Introspection-based Memory De-duplication and Migration

Jui-Hao Chiang
Stony Brook University
Stony Brook, USA
juihaochiang@gmail.com

Han-Lin Li
Industrial Technology Research Institute
Hsinchu, Taiwan
astercase@gmail.com

Tzi-cker Chiueh
Industrial Technology Research Institute
Hsinchu, Taiwan
tcc@itri.org.tw

Abstract

Memory virtualization abstracts a physical machine's memory resource and presents to the virtual machines running on it a piece of physical memory that could be shared, compressed and moved. To optimize the memory resource utilization by fully leveraging the flexibility afforded by memory virtualization, it is essential that the hypervisor have some sense of how the guest VMs use their allocated physical memory. One way to do this is virtual machine introspection (VMI), which interprets byte values in a guest memory space into semantically meaningful data structures. However, identifying a guest VM's memory usage information such as free memory pool is non-trivial. This paper describes a bootstrapping VM introspection technique that could accurately extract free memory pool information from multiple versions of Windows and Linux without kernel version-specific hard-coding, how to apply this technique to improve the efficiency of memory de-duplication and memory state migration, and the resulting improvement in memory de-duplication speed, gain in additional memory pages deduplicated, and reduction in traffic loads associated with memory state migration.

Categories and Subject Descriptors D.4.2 [*Storage Management*]: Main memory

General Terms Design, Management, Measurement

Keywords Memory de-duplication, VM Migration, Introspection

1. Introduction

Memory de-duplication [22] for virtualized servers identifies physical memory pages on a physical machine that are duplicates of each other, consolidates those with identical contents to a single page by pointing these *guest physical pages* to the same *machine physical page*, and marks the machine physical page as read-only. Once any of these guest physical pages is first written, a *copy-on-write* (COW) operation is triggered, which creates a new copy of the machine physical page so as to allow the write to go through. A common way to identify physical memory pages with identical contents is to hash each physical page's contents, compare the resulting hash values, and confirm the equivalence of two pages with the same hash value through a byte-by-byte comparison. Although the above approach has been shown to work empirically, it suffers from the following drawback. Because the page equivalence check is based on hashed value, the above approach cannot establish the equivalence of memory pages whose contents are *don't-cares*, e.g., pages in the free memory pool of the guest OS or application processes. Because the contents of these pages are immaterial, they could have been treated as a zero page without affecting the system's correctness.

VM migration [11] is a powerful building block for improving the availability, power consumption and resource utilization efficiency of virtualized data centers. The bulk of work in moving a VM is the migration of its memory state. Although modern VM migration technology could effectively reduce the connection disruption time of a VM to a sub-second range, it is infrequently triggered in practice because the performance impact of a VM migration transaction on the network is non-negligible due to its bursty nature. One way to mitigate this network performance impact is to avoid migrating unnecessary memory pages, e.g., those that already exist on the target machine or those whose contents are immaterial and thus not worth transferring.

From the above analysis, it is clear that the hypervisor would be able to effectively optimize both memory de-duplication and memory state migration should it have full knowledge of the free memory pool information of every guest VM running on it. For memory de-duplication, the hypervisor could treat free memory pages as zero pages and de-duplicate them accordingly. Moreover, deduplicating these pages does not require page content hashing or byte-by-byte comparison. For memory state migration, the hypervisor could skip transferring free memory pages, and thus cut down the traffic load injected into the network.

One way for the hypervisor to obtain the free memory pool information in the guest kernel is to inject an agent into every guest VM, which at times could be cumbersome and error-prone. In this research, we explored an alternative approach, *virtual machine introspection (VMI)*, which converts memory byte values into semantically meaningful data structures. The accuracy requirement for our introspection technique is higher than those used in security forensics, because our target is a *live* VM and inaccuracy could lead to VM crash. Modern introspection techniques developed for forensics could modify the target VM, including running code inside the target VM where our technique does not have that latitude.

Among all existing VMI tools, most of them are developed based on manual reverse engineering efforts and include hard-coded parsers that are tailored to specific kernel versions. For example, although the base location of the free memory map is available from the debug symbol table for the guest OS and can be manually retrieved, as is done in Xenaccess [13, 17], the internal structures of the free memory map are undocumented and tend to vary from one kernel version to another. For example, the per-page descriptor size has been changed from 24 bytes in the 2006 version of Windows XP to 28 bytes in the 2009 version of Windows XP. We developed a *bootstrapping* VM introspection scheme that could programmat-

VEE'13, March 16–17, 2013, Houston, Texas, USA.

ically identify the free memory pool information of multiple versions of Windows and Linux guests at run time. We believe this makes a novel contribution to the repertoire of techniques used in VMI tool development.

2. Related Work

Without directly hashing the page content, Linux's KSM (Kernel Same-Page Sharing) feature [9] uses the page contents as keys to construct a Red Black tree. Each time a new page is encountered, its content is used as a key to search the tree for a possible match. If a match is found, the new page may be de-duplicated. In addition to storing page contents, the Difference Engine [14] applies an additional *page patching* technique to store some pages as deltas with respect to pre-chosen reference pages.

Satori [16] considers the guest OS's page cache as a promising source for duplicated pages. To avoid periodically scanning the memory pages in guest VMs, it modifies the block device layer of the guest OS (Linux) so that it is *sharing-aware* and enables the Xen hypervisor to intercept all disk accesses. Upon each disk access that populates the page cache, Satori computes the hash values of those new pages that are pushed into the page cache and determines if they are duplicates. In addition, Satori also maintains a list of pages containing replaceable contents, similar to our *don't-care* pages, and provides them to the hypervisor as the new pages used in COW. However, maintaining this list requires modifications to the guest OS, which means that it is not applicable to closed-source OSs.

Besides *memory de-duplication*, there are other approaches to increasing the memory utilization efficiency. IBM's Collaborative Memory Management (CMM) system [21] modifies the guest OS to provide hints to the underlying IBM z/VM hypervisor, so that the hypervisor can page out memory pages of guests, called *host paging*, and use the reclaimed memory space for other purposes. In addition to guest OS modification, this host paging mechanism also requires pausing the guest whose memory is to be paged out, which incurs noticeable performance overhead.

Transcendent [15] modifies the *page cache* implementation of Linux such that the hypervisor provides a free memory pool, called *precache*, that serves as a second-level cache of the guests' page cache. Every time a guest needs to access the disk, it first queries the *precache* to avoid performing any disk I/O operation whenever possible. On the other hand, when a page is evicted out of a guest's page cache, the page is stored in the *precache* for future accesses. The hypervisor could dynamically shrink and expand this *precache* in an on-demand fashion. Because the *precache* is made visible to the guest OS, significant kernel modifications are required, which again limits its applicability.

Virtual machine introspection (VMI) was originally proposed to detect malware intrusion, e.g., kernel rootkits, in guests running on a virtualized server. To narrow the semantic gap between the byte values in a guest's memory pages and the high-level state information they contain, Dolan-Gavitt et al. [12] proposed to log an execution trace of a guest program, extract the instructions from the trace, create a *translated program* from the extracted instructions, and run the resulting program on *Dom0*[1] of Xen outside the monitored guest. This approach requires an OS version-specific step that identifies the instruction sequence to be recorded, extracted and translated, and is not fully automated.

Garfinkel et al. [13] leveraged the *crash* utility [3] to analyze memory pages in Linux guests and used the analysis results for intrusion detection. The *crash* utility was originally used to debug the Linux kernel, and has several hard-coded parts that are specific to each Linux kernel version. Xenaccess [17] was developed recently to extract process and module information of both Windows and Linux guest OS in the Xen hypervisor. However, it could not parse free memory maps, and contains many hard-coded values specific to each guest OS version.

Bryant et al. [10] also proposed to use introspection to identify free memory pages in Linux guests and avoid copying them during VM state cloning. This research gave another example of memory virtualization optimization that benefits from VM introspection. However, this work did not address the issue of how to programmatically extract the free memory map information from different versions and configurations of Linux kernels. Our work does address this issue and the solution is discussed in Section 3.3.

3. Bootstrapping VM Introspection

3.1 Overview

Most existing VM introspection methods rely on manually constructed interpretation programs that extract high-level kernel data structure information from a guest VM's physical address space. The process of building these interpretation programs is usually time-consuming, error-prone, difficult to maintain, and sometimes next to impossible, especially for closed-source OS. One way to avoid these difficulties is to link an agent into the guest kernel when the kernel is built so that the agent could leverage the kernel's symbol and type information to make sense of the guest kernel's address space(e.g., Windows [4], Mac OS X [1]). However, the agent approach is not always possible for proprietary OSs, and entails its own problems for agent update and deployment.

As an alternative to the manual and agent approaches, we have developed two techniques that aim to *programmatically* exploit as much information about a given guest kernel as possible, so as to do away with hard-coded values and parsing logic. For the guest OSs throughout this paper, we focus **only** on the two prevalent ones in large data centers [2], the Windows and Linux OSs. As for other guest OSs, we have not explored and **do not claim** the applicability of VMI techniques to them.

The first technique identifies the type and version of a guest kernel, retrieves the kernel's debugging information either by fetching it from the original software publisher or dynamically building it from the corresponding kernel source code if available, and derives the type and base information of kernel variables and data structures of interest. The second technique takes one step further by incorporating code in the guest OS directly into the interpretation program so as to examine or even manipulate certain guest kernel data structures that are un-documented and even implementation-specific.

These two techniques form a new approach to VM introspection called *Bootstrapping VMI* or BVMI, which, instead of including kernel version-specific hard-coding, dynamically builds up its knowledge about a given guest kernel by fetching and exploiting all available compiler-generated and/or source code information associated with the kernel, and uses this knowledge to make sense of the bytes values in the guest memory space in a way that approximates the effectiveness of the agent approach. In the following two subsections, we describe how we applied these two techniques to successfully identifying the free memory pool of various versions of Windows and Linux OSs.

3.2 Free Memory Pool Identification for Windows

On Windows, executable files, dynamically linked libraries (DLLs), and kernel images are all in the Portable Executable or PE format [8]. When a PE file is loaded into memory, it is stored at a starting address known as the *base address*. The virtual address of every symbol in a PE file is represented as an offset with respect to the file's base address, also known as the Relative Virtual Address

[1] Dom0 is the privileged VM in Xen while other VMs are called DomU.

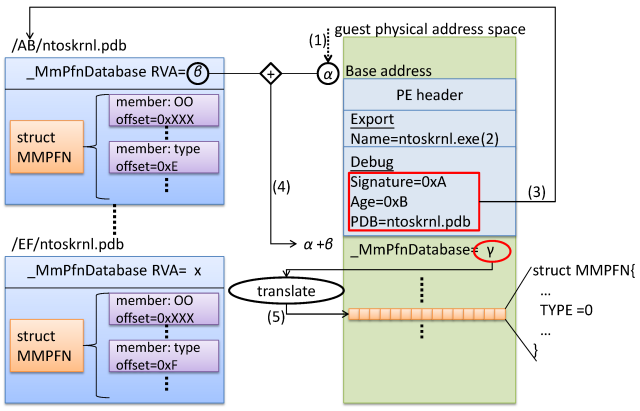

Figure 1. *Introspecting a Windows virtual machine requires matching a a Windows kernel PE file loaded into a guest VM (on the right) with its corresponding debugging PDB file fetched from Microsoft's Symbol Server (on the left).*

(RVA). Therefore, a symbol's absolute virtual address is equal to the sum of its RVA and the *base address* of where the associated PE file is loaded. As shown on the right side of Figure 1, when a PE file is loaded, its headers are also loaded. The two items in the PE header that are relevant here are *export data directory*, which contains the name of the current PE file, e.g., `ntoskrnl.exe` for the Windows kernel [20], and *debug data directory*, which contains a 128-bit *Signature* and a 32-bit *Age* to uniquely identify the PE file's associated debugging information.

For each PE file, the linker, e.g., Microsoft Visual C++, may optionally produce a PDB file that contains the associated debug information, such as the symbol table. For widely used PE files, Microsoft publishes their PDB files on its public web site. The PDB file by default contains the name, address, and type information for functions and variables, e.g., the RVA of a static variable declared in the program. To associate a PE file with its PDB file, both of them include a unique combination of *Signature* and *Age* values. In addition, Microsoft provides a Debug Interface Access (DIA) SDK [4] that allows developers to extract detailed debug information about a PE file, e.g., one can use a depth-first-search algorithm to query a specific member of a user-defined structure, because the latter is stored as a tree hierarchy inside its PDB file.

The *memory map* in a Windows kernel is called the PFN (Physical Frame Number) Database, which is a statically allocated array of page descriptors, each of type `struct _MMPFN`, and is located in a contiguous region of the guest physical address space [20]. The symbol name of the PFN Database is `_MmPfnDatabase`.

After a Windows OS boots up, all its free pages are in the *Zeroed* page list, which is the free memory pool every page in which is all-zero. When a free page is allocated to a process, it is taken out of the *Zeroed* page list. Upon the exit of a process, pages in its working set are returned to the *Free* page list. For security reasons, the Windows OS uses a *zero page thread* to zero out the pages in *Free* list and put them into the *Zeroed* page list for future reuse.

Based on the above information, we developed a BVMI program[2] to identify the free memory pool of multiple versions of the Windows OS. The BVMI program considers a guest page is free if it is located in the *Zeroed* list, i.e., when the *type* of the page descriptor is equal to `ZeroedPageList`, which is defined as a member of an enumeration type `_MMLISTS`. This rule applies to all Windows OS versions ranging from Windows XP to the latest Windows 7 [18–20]. However, because the detailed layout of the enumeration type `_MMLISTS` may vary from kernel version to kernel version, we have to rely on kernel version-specific PDB files to help determine the exact layout of this data structure and use this knowledge to perform free memory page check.

To summarize, the BVMI program takes the following steps to programmatically traverse the PFN database and inspect each page descriptor, as outlined in Figure 1.

(1) Given a PE file, BVMI first scans its export data directory to confirm its name is indeed `ntoskrnl.exe` (a Windows kernel), and then scans the remaining file for a page containing the magic string "MZ", whose base address is the guest kernel's base address, $Base_{kernel}$.

(2) BVMI extracts the *Signature* and *Age* fields from the PE file's debug data directory, and uses them to construct an URL that can be used to access the PE file's PDB file from Microsoft's public symbol server.

After the kernel image's PDB file is downloaded, BVMI uses the DIA SDK to search for the relative virtual address of the PFN database, which is denoted as $Base_{PFN}$, and to traverse the `_MMPFN` structure to derive the data structure's size and the offset of its member *type*.

(3) BVMI computes the start physical address of the PFN database by adding $Base_{PFN}$ to $Base_{kernel}$, translates it to its corresponding machine physical address, maps the array into its own virtual address space, and examines the $type$ field of every PFN database entry to determine if the corresponding physical page is free using the layout information extracted from the `_MMPFN` structure.

3.3 Free Memory Pool Identification for Linux

When a Linux kernel image is built, a special file called *System.map* is generated, which contains a mapping between the symbol names of all exported variables and functions in the kernel source and their absolute virtual addresses. A Linux kernel image contains a real-mode kernel image, whose header contains detailed Linux version information such as the kernel version, distribution, and build timestamp, which can be used to track down the corresponding distribution source package.

To avoid memory fragmentation, Linux uses a buddy system memory allocator, which organizes free memory pages into groups of physically contiguous pages, each with a size that is a power of two. The first page of every free memory page group is called a *Buddy page*, which represents the entire group and uses the *private* field of its page descriptor to record the number of physically contiguous pages in its group. For example, if the *private* field of a *Buddy page*'s descriptor is n, there are 2^n contiguous free pages in its group. Therefore, one can uncover free pages in a guest OS by first identifying Buddy pages and then all other pages in their groups. Linux supports two memory models: (1) a *flat* model using a one-dimensional memory map, and (2) a *sparse* model using a two-dimensional memory map. The exact memory model used in a kernel is selected at the kernel configuration step, and remains fixed after the kernel is built.

For the same kernel source, each Linux distribution, e.g., Ubuntu, maintains its installation package for kernel image, which is usually stripped off kernel debugging information. One could recreate the debug information associated with a kernel distribution, by downloading the distribution's corresponding kernel source and building a kernel image with the `CONFIG_DEBUG_INFO` option enabled from it.

The BVMI program for Linux is more difficult to develop than that for Windows, because the Linux kernel is highly configurable,

[2] For implementation, it runs in the Dom0 domain of Xen, which has the privilege to access all memory pages of guest VMs.

and the same kernel source could be compiled into kernel versions with very different structures and configurations. For example, configuration primitives such as `#ifdef` could conditionally trigger different compiler pre-processing and post the following issues. First, the memory model configuration option determines whether the memory map is represented as a one-dimensional or two-dimensional array. Second, size and layout of the page descriptor data structure may vary from one kernel version to another. Third, semantics associated with data structure values used to identify Buddy pages vary from one kernel version to another.

The first two issues could be resolved by using the GDB interface to query the debug information associated with the input kernel image. However, the last issue could not be easily resolved because which data structure field is used to identify Buddy pages changes from one kernel version to another. For example, in Linux versions older than 2.6.18, a page is said to be a Buddy page if the 19th bit of the `flags` field in the page's descriptor is set, but in version 2.6.38, it uses the `_mapcount` field.

Fortunately, across all Linux kernel versions, there is a standard inline function, `PageBuddy(struct page*)`, that takes a page descriptor structure as the input and returns true if the page is a Buddy page. However, the implementation of this function is different for different kernel versions. One way to solve the Buddy page identification problem is to incorporate the logic of the PageBuddy function for different Linux kernel versions into the BVMI program so as to identify free memory pages in guest OSs that run different versions of Linux. However, this approach is cumbersome and error-prone. Instead, we directly leverage the `PageBuddy` function's implementation in each Linux kernel version by calling it from the BVMI program. More concretely, we wrote a stub function, called `GFN_is_Buddy`, which determines whether a given GFN in a guest is a Buddy page or not by calling the guest's `PageBuddy` inline function. The input argument of this stub function is a GFN in a guest, and this interface is the same across all Linux kernel versions.

```
/* stub.c: guest kernel module */
int GFN_is_Buddy(unsigned long GFN)
{
    struct page *page;
#ifdef CONFIG_FLATMEM
    /* Flat model: use map as 1D array */
    page = (struct page*)mem_map[GFN];
    return PageBuddy(page);
#else
    ...
}
```

Given a guest VM using a specific Linux kernel version, we dynamically compiled this stub function against the kernel version's source code and configuration, and produced an ELF file, `stub.o`. Then we linked this `stub.o` file with the BVMI program so that the latter can call the `GFN_is_Buddy` function on every guest page descriptor it examines. We believe the above approach represents a new VM introspection technique in that it pioneers the use of kernel version-specific code to interpret kernel version-specific undocumented data structures. Although this technique needs access to the kernel source code and configuration file used by a Linux guest, it does not require any knowledge of the detailed layout or semantic information associated with any kernel data structures.

The following code snippet shows the independent free page check program (`FreePageCheck.c`) that includes the previous `GFN_is_Buddy` function and services requests from the BVMI program. The communications between the BVMI program and the free page check program is based on a shared memory mechanism.

```
/* FreePageCheck.c: kernel version-specific
 * code to check if a page is a Buddy page */
```

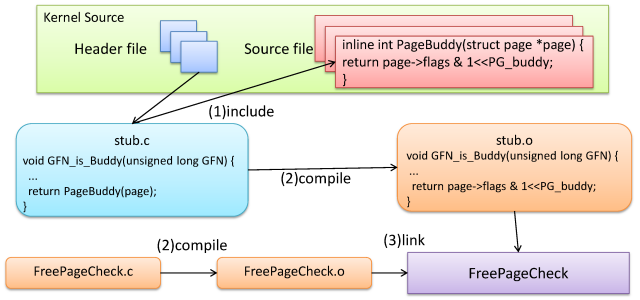

Figure 2. *Compiling the `GFN_is_Buddy` function against the source code and configuration file of a guest's Linux kernel version to generate a `stub.o` file, and linking it with `FreePageCheck.c` to form the independent free page check program.*

```
extern int GFN_is_Buddy(unsigned long);
void check_free_page(int *free_map)
{
  for (gfn = 0; gfn < guest_max_gfn; gfn++)
    if (GFN_is_Buddy(gfn)) free_map[gfn] = 1;
}

int main()
{
  check_free_page(free_map);
  /* Share free_map with BVMI program */
}
```

Figure 2 shows the steps taken to generate a kernel version-specific `stub.o` and link it with the compiled ELF object of `FreePageCheck.c` to form a kernel version-specific free page check program, which could answer queries from the BVMI program on whether certain pages are free. Note that the memory model is also determined in the stub.o after compiled with the guest kernel source. In summary, the steps to identify free memory pages in a Linux-based guest VM are

(1) Searches the header of Linux guest real-mode kernel image for the magic string "HdrS", then extracts the kernel version information, and uses the kernel version information to either retrieve the corresponding free page check program if it exists, or dynamically composes the corresponding free page check program based on the kernel version's source code and configuration file,

(2) Extracts from the `System.map` file associated with the guest kernel's version the location of the memory map, and

(3) Traverses every page descriptor in the memory map by calling the free page check program to determine if each traversed page is a Buddy page.

3.4 Discussions

The two techniques described in this section still require a prior knowledge of how to link high-level information being sought after (e.g. memory page status) with specific kernel data structures (e.g. memory map), and thus manual efforts to build the introspection mechanism. In addition, because a kernel image is a product of the kernel source code, the configuration file, and the compiler, these two techniques may fail to deliver correct introspection when any change is made to the source code, the configuration file, or the compiler used to build a guest kernel.

The second technique is an instance of exploiting a guest OS's own code to help the VMI program to make sense of its own byte sequences [12]. While conceptually simple, it has two potential is-

sues. First, the guest OS functions used to interpret the guest physical address space must be self-contained and do not reference any other kernel variables or functions. Second, as the interpretation program inspects a guest's physical address space, the guest's state must not go through any modification; otherwise the interpretation program may break because of dangling pointers, e.g., a next pointer in a linked list item becomes invalid as a result of guest state modification. Based on our prototyping and test experiences, neither of these two issues becomes a real problem.

The principles underlying the proposed introspection technique are generic, and its applications to different versions of guest OS vary from one guest OS to another. With the help of these principles, it takes a much smaller amount of effort to custom-build a free-memory-pool introspection technique for a new version of guest OS than without such principles.

One may worry the applicability of such technique when VM uses highly customized guest OSs. However, the most common service scenario for IaaS (Infrastructure as a Service) is for the IaaS provider, e.g., Amazon [2], to supply a set of pre-prepared VM images for users to start their VMs, and users are discouraged from modifying their VMs' OSs. Therefore, the proposed introspection technique is a good fit with the prevailing IaaS service model, as well as PaaS (Platform as a Service) offerings, in which users do not get to choose VM images.

4. Generalized Memory de-duplication

We built a Generalized Memory de-duplication (GMD) engine that leverages the free memory pool information in guest VMs to de-duplicate pages that have identical or don't-care contents. This GMD prototype is based on the Xen hypervisor. In Xen, the *guest virtual* addresses in a VM are translated to their *guest physical* addresses, and then to *machine physical* addresses, i.e., which are the actual the physical addresses used to access the memory. Accordingly, the Machine Frame Number (**MFN**) is a page number in the machine physical address space whereas the Guest Frame Number (**GFN**) is a page frame number in the guest physical address space. Normally, each GFN of a guest OS is mapped to a unique MFN allocated by the hypervisor. To increase the memory utilization efficiency, Xen supports a *memory sharing* mechanism similar to that for sharing memory among processes in a conventional OS, the COW mechanism, i.e., all processes map a shared page as read-only; when a process Z first writes the shared page, a write-protection fault occurs and OS allocates and maps a new memory page for Z, which is marked as read-writable and initialized with the faulted page's content. In Xen, this is achieved by the $unshare$ memory sharing API function.

Two other important API functions for memory sharing are $nominate$ and $share$. The first function accepts a VM identifier and a GFN as input parameters, and marks the corresponding page as read-only with a returned $handle$, which uniquely identifies the page. Accesses to such a $handle$ are protected by a global exclusive lock. The second function takes two $handles$ as input parameters. If both $handles$ are valid, Xen maps the two GFNs associated with these two $handles$ to the MFN associated with the first $handle$, and thus frees the physical page corresponding to the MFN associated with the second $handle$. In addition to $nominate$ and $share$, Xen also provides API calls that allow the *Dom0* kernel to map a GFN of a DomU VM into the virtual address space of a user-level program running on it, and to $translate$ a given guest virtual address to its corresponding guest physical address.

The proposed GMD engine leverages the free memory pool information about guest VMs, treats the free memory pages as duplicates of an all-zero page, and de-duplicates them accordingly. The current GMD engine prototype is implemented as a user-level program that runs in *Dom0* and implements de-duplication using

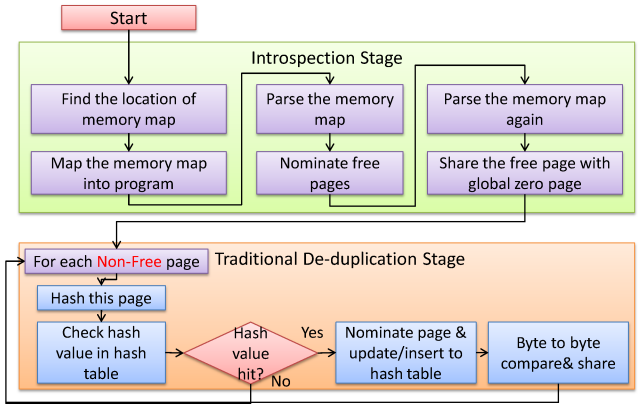

Figure 3. *The workflow of the proposed Generalized Memory de-duplication (GMD) engine, which comprises two stages: the* Introspection *stage to identify free memory page in guest VMs and the* de-duplication *stage that de-duplicates pages using hashing and byte-by-byte comparison.*

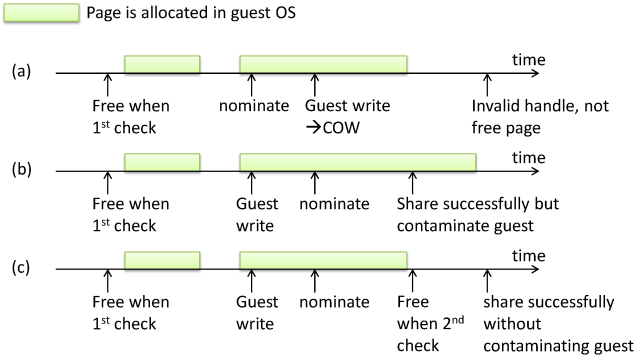

Figure 4. *The GMD engine checks if a guest page is free twice to avoid a race condition illustrated in (b), where a page is detected free, modified by the guest, and then nominated and shared by the GMD engine. Checking pages are free twice avoids the data corruption problem due to this race condition, as shown in (c).*

the Xen hypervisor's primitives mentioned above. As shown in Figure 3, the GMD engine consists of two stages: the *Introspection* stage and the *de-duplication* stage. In the *Introspection* stage, the GMD engine first calls the BVMI program to identify and record free guest pages into a bitmap, $free\_map1$, and nominates them, then walks the memory map again to check if those pages in $free\_map1$ are still free and marks those that are still free in another bitmap, $free\_map2$, and finally shares each page is that is in both $free\_map1$ and $free\_map2$ with the all-zero page without comparison.

In the *de-duplication* stage, for each non-free page P, the GMD engine computes a hash value using its page contents, looks up the resulting hash value in a global page content hash database, and if a hit is found, nominates P and shares P with the hit page. If the byte-by-byte comparison comes back with the match, the $share$ function removes duplicate page and returns success.

The GMD engine checks *twice* if a memory page in a guest OS is free during the Introspection stage, because the GMD is implemented as a user-level program running in Dom0 and there is a potential race condition between when it detects a guest memory

page is free and when it nominates the page for sharing. As shown in Figure 4(a), if a guest page that is identified as a free page is allocated and modified after it is nominated, the modification triggers the COW mechanism, which in turn destroys the page's $handle$ returned by the $nominate$ call, and the following $share$ call will abort because the $handle$ is invalid. If, however, the modification appears after the first free page check but before the $nominate$ call, as shown in Figure 4(b), the page's $handle$ continues to be valid when the GMD engine nominates and shares the page with the all-zero page, and eventually the modification is lost. Instead, we solve this problem by checking free memory pages twice. With a second free page check shown in Figure 4(c), a guest page can only be nominated and shared only if the second check also reports it is free. If a guest page is still free in the second check, it means the page could be safely de-duplicated with the all-zero page because its contents can be thrown away.

Proof For guest OS, we assume it performs the following two steps when allocating a memory page: (1) Mark the memory page as non-free (2) Give the page to the user or kernel component, which then stores information into it later. The dual-check mechanism includes three operations $C1$ (first check if the page is free), R (mark the page as read-only), and $C2$ (second check if the page is free). Only after $C2$ is successful will a page be mapped to a zero page. In the following, $X < Y$ means X happens before Y. Thus, it is true that $C1 < R < C2$ and $(1) < (2)$.

A race condition that could lead to data corruption must satisfy the following condition: $C1 < (1) < (2) < R$, i.e., the guest allocates the page after $C1$ and then modifies it before R. In this case, the COW mechanism does not have a chance to get triggered and the page may be corrupted if $C2$ is not included. However, after $C2$ is included, the proposed scheme could effectively detect this race condition and abort the attempt to reclaim the page, because $C2$ would find the page no longer free and thus won't map it to a zero page.

5. Free Memory Pages-Avoiding VM Migration

A major performance metric for VM migration is its impact on the network due to memory state transfer. Because the contents of the free memory pages of a VM are don't-cares, they do not need to be moved when the VM is migrated. Avoiding transferring free memory pages of a migrated VM is thus an effective way to reduce the network performance impact of a VM migration transaction.

Xen uses an iterative memory state transfer scheme by organizing each migrated VM's memory into chunks of 1024 pages. In the first iteration, for each chunk, Xen first sends to the target machine a map, *pfn_type*, each entry of which describes the type of each transferred memory page, e.g., *invalid* or *regular* data page, and then sends the contents of all valid pages in the current chunk. To avoid transferring free memory pages to the target machine, we introduced one more type, *free*, to denote pages that are valid but free. By consulting with the BVMI program, Xen on the source machine identifies the guest physical pages in a migrated VM that are free, marks the corresponding entries in the *pfn_type* map as *free*, and skips transferring them to the target machine. For all free pages of a migrated VM, as indicated in the received *pfn_type* map, Xen on the target machine de-duplicates them to an all-zero page. For the remaining iterations, Xen does not leverage introspection, but focuses only on the transfer of dirtied pages.

6. Performance Evaluation

The test machine used in this study contains an Intel Xeon E5640 quad-core processor with VT and EPT enabled, 24 GB physical memory, and a 500 GB hard disk. The host runs Xen-4.1 with CentOS-5.5 as the *Dom0* kernel. All our VMs are configured with 1 virtual CPU and 4 GB memory, which corresponds to something between the *Small Instance* and *Large Instance* classes of Amazon's Elastic Compute Cloud (EC2) service [2] and should be representative for normal server or desktop applications.

We tried guest VMs running 32-bit and 64-bit versions of Windows and Linux, including **Win7-64** (64-bit Windows 7), **WinXP-32** (32-bit Windows XP with Service Pack 2 installed), **Centos-64** (64-bit Centos 5.6 with the 2.6.18 Linux kernel and Sparse memory model configured), and **Debian-32** (32-bit Debian-6.0.2.1-i386 with the 2.6.32 Linux kernel and Flat memory model configured). As for input workloads, we ran the following three benchmarks inside the guest VMs:

- **Video-Creation**, **E-Learning**, and **Office**:
  Three workloads from SYSmark2007 [7] which simulates the three different classes of business user behaviors on Windows desktop machines.
- **Banking**, **Ecommerce**, and **Support**:
  Three workloads from Specweb2009 [6] which are designed to evaluate web server performance.
- **Specjbb**: Specjbb2005 [5]. A SPEC benchmark that emulates a three-tier client/server system and is designed to evaluate the performance of server-side Java applications.

While Sysmark is a stand-alone benchmark that runs on Windows guests, Specweb is used to generate workloads from simulated clients and require running a web server on a Linux-based guest VM. As for Specjbb, we apply it to both Windows and Linux guest OSs.

Table 1 show the percentage of free pages in the 4 test VMs running four different workloads when the free memory pool grows and shrinks during the test period. The results in these tables present the characteristics of the workloads running inside the test VMs. For memory de-duplication, we expect the average shared pages by introspection mechanism is proportional to the average percentage of free pages. As for VM migration, we also expect the percentage of skipped transferred pages by introspection to approximate the average percentage of free pages.

6.1 Memory de-duplication

We used three metrics to evaluate the effectiveness of a memory de-duplication scheme: (1) the number of pages it reclaims, (2) the performance overhead it incurs, and (3) the performance penalty it imposes on guest VMs. The performance overhead of conventional memory de-duplication schemes mainly comes from hash computation and byte-by-byte comparison, whereas that for the proposed introspection-based memory de-duplication approach arises from introspection, (2) and (3) are different for two reasons. First, one could minimize performance impacts on guest VMs by carefully scheduling memory de-duplication operations when the CPUs are less loaded. Second, guest VMs may encounter additional protection faults, context switches and hypercalls ($unshare$ in the case of Xen) as a result of writes to pages that are protected by the copy-on-write mechanism. Therefore, a guest VM's run-time performance penalty depends on the number of $unshare$ calls it triggers.

To isolate the performance benefit of memory de-duplication schemes based on hashing and introspection, we compare the following four configurations of the GMD engine, using the **Baseline** configuration as the basis of effectiveness calculation:

- **Baseline**: The GMD engine is totally turned off, no memory pages are de-duplicated and no memory de-duplication overhead is incurred.

	E-learning	Video-Creation	Office	Specjbb
Win7-64	77, 52, 2	75, 40, 4	69, 61, 57	84, 73, 69
WinXP-32	78, 58, 17	77, 43, 0	70, 65, 53	86, 72, 9
	Banking	Ecommerce	Support	Specjbb
Centos-64	93, 92, 90	93, 92, 91	92, 76, 69	91, 81, 77
Debian-32	92, 91, 90	92, 91, 90	93, 75, 68	92, 83, 79

Table 1. Percentage of free pages against the test VM's total memory size for four test VMs each under four different workloads where each grid shows the maximum, average, and minimum value.

- **Intro**: Only the *Introspection* stage of the GMD engine is turned on.
- **Dedup**: Only the *de-duplication* stage of the GMD engine is turned on.
- **IntroDedup**: Both stages of the GMD engine are enabled. Free pages are de-duplicated by the Introspection stage and non-free pages are de-duplicated by the de-duplication stage.

In this study, we mainly focused on the memory de-duplication within an individual VM, and ignore the inter-VM memory de-duplication, because traditional content-based de-duplication already does a credible job at removing inter-VM duplicates. As a result, in each experiment, we ran only one VM, on which a particular input application workload is run. Because the GMD engine takes less than one minute to complete one de-duplication round through a VM with 4GB of physical memory, we configured the GMD engine to run once every minute by default. We also varied the invocation frequency of the engine to explore the trade-off between the cost and gain of memory de-duplication.

6.1.1 Effectiveness of Introspection for Windows

Figure 5 shows the comparison of Win7-64 VM among the three GMD configurations, $Intro$, $Dedup$ as well as $IntroDedup$, in terms of memory saved, de-duplication overhead, and performance impacts on guest VMs, under four different input workloads. Because the Windows OS zeros out a memory page before putting it in the free memory pool, $Dedup$ can de-duplicate any free page that $Intro$ can de-duplicate. So in theory, the amount of memory saved by $Dedup$ should be larger than that by $Intro$, and is equal to that by $IntroDedup$, as shown in Figure 5(a), where the metric is the percentage of the test guest VM's physical memory that is de-duplicated and shared by the GMD engine at the end of each minute during the experiment run.

However, for the E-Learning workload, the amount of memory shared by $Dedup$ is smaller rather than larger than that by $Intro$, and for the Video-Creation workload, the amount of memory shared by $Dedup$ is smaller than rather than equal to that by $IntroDedup$. These anomalies arise mainly because the amount of time and work required to perform one memory de-duplication round through the test VM's physical memory space is different for these three configurations, as shown in Figure 5(b). As expected, $Dedup$ is the most time-consuming because it needs to perform per-page content hashing and byte-by-byte comparison, $Intro$ is the quickest because it only needs to examine specific guest kernel data structures, and $IntroDedup$ is between the two extremes because it is a hybrid of $Intro$ and $Dedup$.

Figure 5(c) shows the average percentage of total memory pages that are $unshared$ each minute by the test VM under the four workloads, and mirrors the results in Figure 5(a), because the number of pages "unshared" is directly correlated with the number of pages that were previously shared by the GMD engine and later allocated and modified. Two factors affect the performance degradation of the test VM when memory de-duplication runs in the background: (1) the overhead of the GMD engine's own de-duplication operations, e.g., hashing computation and locking of

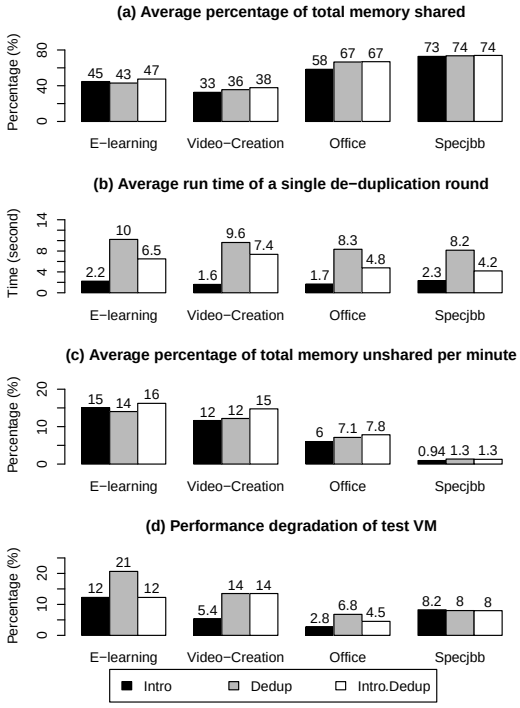

Figure 5. *Comparisons among $Intro$, $Dedup$ and $IntroDedup$ under four different workloads running on a Win7-64 test VM with 4GB physical memory in terms of (a) the average percentage of total memory shared by the GMD engine per minute, (b) the average time required to perform a single memory de-duplication round through the test VM's physical memory space, (c) the average percentage of total memory unshared by the test VM per minute, and (d) the performance penalty experienced by the test VM.*

memory pages, and (2) the number of copy-on-write exceptions because of the VM's writes to shared pages. Therefore, the test VM's performance degradations shown in Figure 5(d) reflect the combined effects in Figure 5(b) and Figure 5(c). Because the differences in the number of unshared pages among the three GMD configurations are small, the performance degradation is influenced more by the de-duplication overhead than by the amount of memory unsharing. Among the three configurations, $Intro$ imposes the minimum performance penalty on the test VM. As for WinXP-32, the results are similar as Figure 6 shows. The overall performance degradation may be a little bit high at first sight. One major reason is that the current scanning frequency, e.g., every one minute, could be high comparing to other research work, e.g., the default scan frequency of VMware is once every 60 minutes, and we will look into this issue in Section 6.1.3.

The other way to evaluate the effectiveness of $Intro$ is to analyze the relationship between the average percentage of free

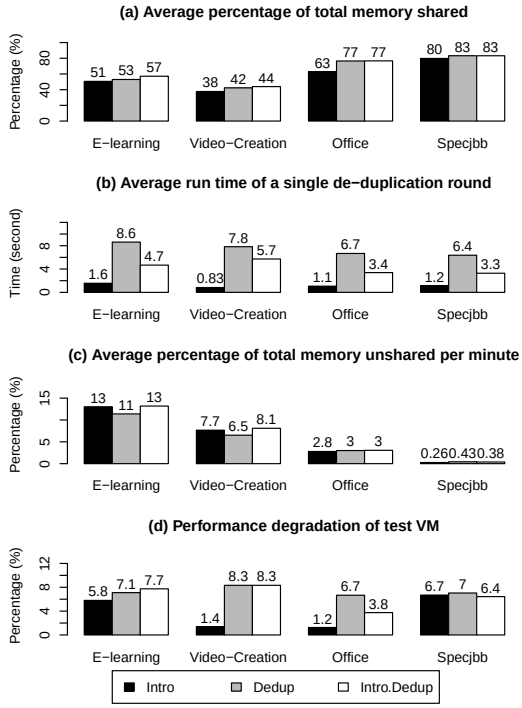

Figure 6. *Comparisons among $Intro$, $Dedup$ and $IntroDedup$ under four different workloads running on a WinXP-32 test VM with 4GB physical memory in terms of the same set of metrics as in Figure 5.*

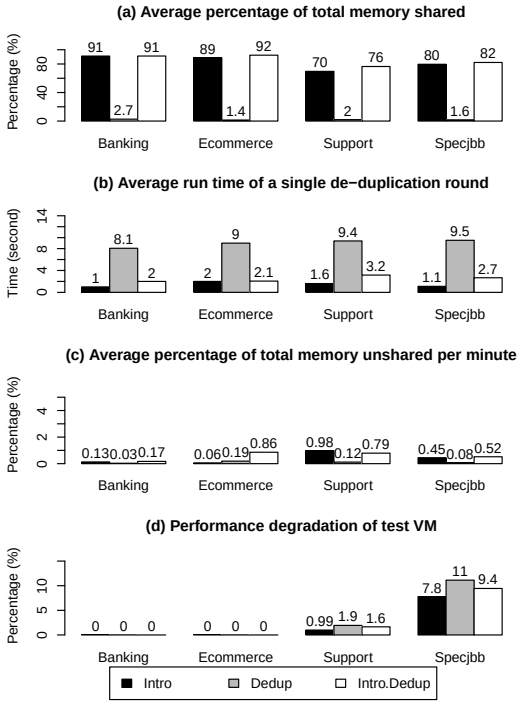

Figure 7. *Comparisons among $Intro$, $Dedup$ and $IntroDedup$ under four different workloads running on a Centos-64 test VM with 4GB physical memory in terms of the same set of metrics as in Figure 5.*

pages and shared pages. If we subtract the average percentage of unshared pages from the average percentage of free pages, we expect the result to be proportional to the average percentage of shared pages. Taking Win7-64 VM as an example, the result of subtraction referring to Table 1 and Figure 5 is 36%, 28%, 54%, and 72% for the four workloads, which matches the trend of the average percentage of shared pages. The same kind of analysis can be applied to all other test VMs, which we will not repeat later due to redundancy.

In summary, on the Windows platform, most deduplicable pages within an individual VM are free memory pages that are zeroed; as a result, $Intro$ is able to discover the majority of the deduplicable pages that $Dedup$ can, and the marginal value provided by the vanilla de-duplication stage in $IntroDedup$ is relatively minor. On the other hand, the de-duplication overhead of $Intro$ is on average four times smaller than $Dedup$'s. In terms of performance impacts on the test VM, $Intro$'s is also significantly smaller than $Dedup$'s. Therefore, as far as intra-VM memory de-duplication for Windows guests is concerned, $Intro$ is the clear choice.

6.1.2 Effectiveness of Introspection for Linux

Unlike Windows, the Linux kernel does not zero out a memory page to be freed before putting it in the free memory pool. Consequently, traditional memory de-duplication schemes cannot easily identify these pages and de-duplicate them, whereas the proposed GMD engine can. To ensure that free memory pages are not zero, we wrote a program to allocate as many free memory pages as possible, write random contents to them and then free all of them, before running any experiments. Without this tweak, our experiment results show what the $Intro$ mechanism on Linux VMs still holds the same advantage as the results from Windows VMs in terms of effectiveness of de-duplication and performance impact on guests.

As shown in Figures 7(a) and 8(a), $Dedup$ can barely de-duplicate any memory page, whereas by leveraging free memory map information, $Intro$ can de-duplicate most of the free memory pages. Moreover, the run time of $Intro$ is significantly lower than that of $Dedup$, because the latter blindly computes hash values of all guest physical pages, as shown in Figures 7(b) and 8(b). For the same reason, the marginal value of the de-duplication stage of $IntroDedup$ is also small when compared with $Intro$. Other than the above, the conclusions drawn from Figures 7 and 8 are similar to those drawn from Figure 5. One notable result is that the performance degradation when the test VM runs Specweb is close to zero, as shown in Figures 7(d) and 8(d). This is because the memory usage of Specweb is pretty static and hence not much memory unsharing takes place at run time.

6.1.3 Minimizing Performance Degradation

Ideally, the performance impact of a memory de-duplication scheme on the test VM should be minimal, preferably close to zero. Unfortunately, the performance degradations shown in Figures 5(d), 6(d), 7(d) and 8(d) are non-trivial. If the memory usage of a test VM is fluctuating dynamically, the free memory pages reclaimed by the GMD engine at one point are likely to be subsequently returned to the test VM when it allocates from the free memory pool. In such a case, the effort of sharing memory pages is a waste; worse yet, the test VM experiences additional performance penalty because of copy-on-write exceptions. Figure 9 shows the average amount of memory shared and the performance degradation of $Intro$ when the test VM runs the four workloads on Win7-64 and when the invocation frequency is once per minute, per two minutes, per four

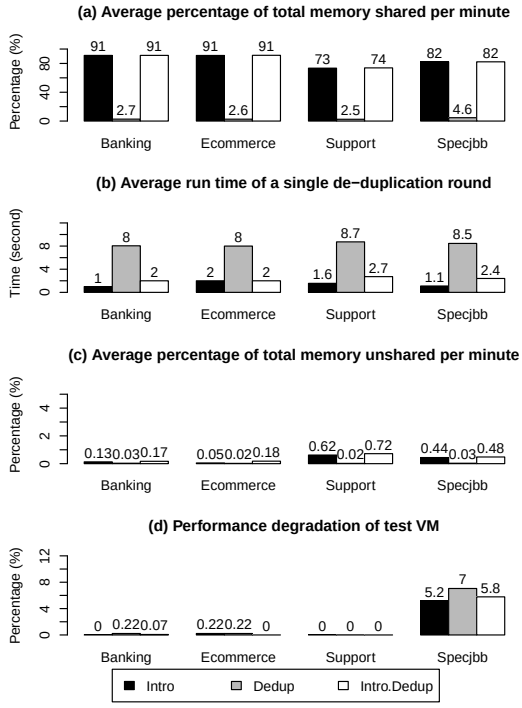

Figure 8. *Comparisons among $Intro$, $Dedup$ and $IntroDedup$ under four different workloads running on a Debian-32 test VM with 4GB physical memory in terms of the same set of metrics as in Figure 5.*

minutes and per eight minutes. As expected, the more frequently the $Intro$ GMD engine runs, the more memory pages it could de-duplicate, and the higher the performance degradation[3]. Unfortunately, while decreasing the de-duplication frequency decreases the amount of memory shared, it does not seem to be able to cut the performance degradation to 0%. This is because the current implementation of $Intro$ shares *all* free pages, and the test VM is bound to encounter some copy-on-write exceptions when it allocates additional memory pages.

6.2 Memory State Migration

We used two metrics to evaluate the effectiveness of a VM migration scheme: (1) the amount network traffic injected by a VM migration transaction, and (2) the total VM migration time. Conventionally, the amount of network traffic injected by a VM migration transaction is directly proportional to the total memory size of the migrated VM, i.e., 4GB in our test setup. It takes roughly 40 seconds to transfer this VM's memory state on our Gigabit Ethernet-based testbed, whose TCP throughput is 819.2 Mbps. By leveraging BVMI to identify free memory pages in a VM that is to be migrated, the hypervisor could skip transferring these free memory pages during the VM migration transaction, and this significantly cuts down the migration-induced network traffic volume.

To assess the performance benefit of introspection-based VM migration, we compare the following two VM migration schemes using the above two metrics:

[3] Because Specjbb runs inside a JVM, its execution does not directly affect the memory footprint and the number of unshared pages in the underlying guest. Consequently, varying the invocation frequency of GMD has little impact on Specjbb's performance.

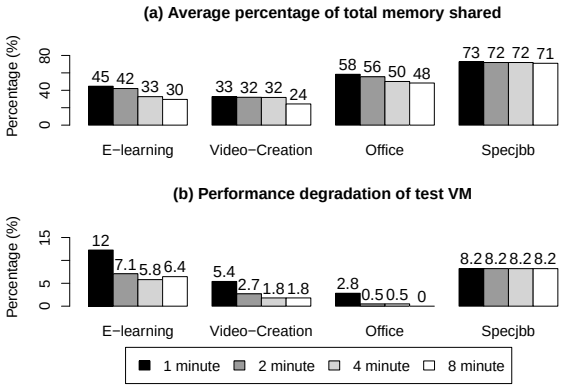

Figure 9. *Impacts of the invocation frequency of $Intro$ (once every 1 minute, 2 minutes, 4 minutes and 8 minutes) on the average amount of memory shared and the performance degradation of the test VM when it runs the four test workloads on Win7-64.*

- **BaseMigrate**: The conventional VM migration scheme implemented in Xen.
- **IntroMigrate**: BaseMigrate with the optimization that avoids transferring free memory pages as identified via VM introspection.

In each run, we triggered a migration of the test VM at a randomly chosen time and measured the network traffic load and migration time, and reported the average of the measurements of multiple runs. Due to space constraint, we only present the results of two types of test VMs, Win7-64 and Debian-32, because WinXP-32 and Centos-64 have similar results.

Figure 10(a) compares the injected network traffic volumes of $BaseMigrate$ and $IntroMigrate$ for a Win7-64 VM under four different test workloads. Compared with $BaseMigrate$, $IntroMigrate$ reduces the network traffic in the first iteration of memory state transfer, depicted as "1st-Iteration" in the figure, by 48%, 41%, 62%, and 81% for E-learning, Video-Creation, Office, and Specjbb, respectively. As expected, the percentage of network traffic reduction is roughly proportional to the average percentage of free pages shown in Table 1, because information extracted by introspection is used only in the first iteration.

For the remaining iterations of memory state transfer, depicted as "Remaining" in the figure[4], surprisingly $IntroMigrate$ also cuts down the network traffic volume by 8%, 57%, 75%, and 9% for E-learning, Video-Creation, Office, and Specjbb, respectively, even though no introspection-derived information is used in these iterations. This reduction originates from the fact that when the first iteration is shortened, fewer memory pages are dirtied in the first iteration, the second iteration is also shortened, even fewer pages are dirtied in the second iteration, and so on. The introspection benefit to the remaining iterations is apparent for the Video-Creation and Office workload, but not so obvious for E-learning because the network traffic due to the remaining iterations is small to begin with, and for Specjbb because it is memory-intensive and introduces a large number of dirtied pages in the remaining iterations regardless of the length of the first iteration.

Figure 10(b) shows that $IntroMigrate$ cuts down the total migration time from 40, 44, 52, and 56 seconds to 25, 29, 21, and 30 seconds, or by 38%, 34%, 60%, and 46% for E-learning,

[4] The CPU and I/O states are only a few KBytes and thus ignored in this discussion.

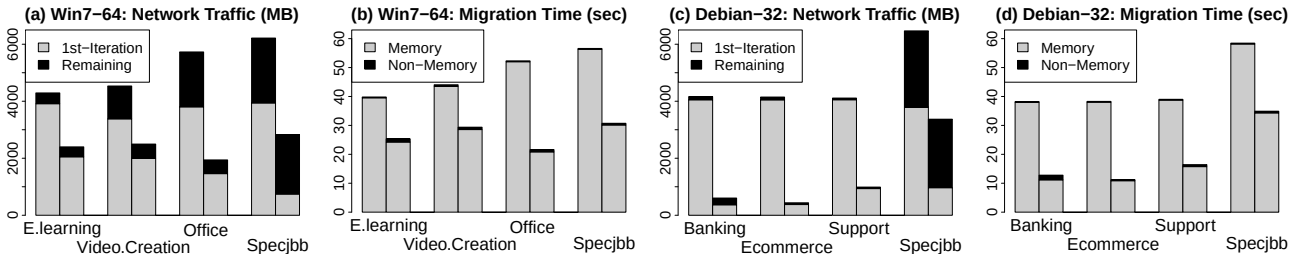

Figure 10. *Comparison of injected network traffic volume and total migration time between $BaseMigrate$ and $IntroMigrate$ for four different workloads running on a Win7-64 VM and a Debian-32 VM. For each workload, the left and right bar represent the result of $BaseMigrate$ and $IntroMigrate$ respectively. In subfigure (a) and (c), "1st-Iteration" and "Remaining" correspond to the injected network traffic volume in the first and remaining iterations during a VM migration transaction, respectively. In subfigure (b) and (d), "Memory" and "Non-memory" correspond to the memory state migration time and the migration time for other VM states, respectively.*

Video-Creation, Office, and Specjbb, respectively. The amount of migration time reduction is proportional to the amount of reduced network traffic or memory state transfer, which accounts for more than 96% of the total migration time. The non-memory portion of the migration time is too small to be noticeable.

Figure 10(c) and 10(d) show similar benefits for the Debian-32 test VM. When compared with $BaseMigrate$, $IntroMigrate$ reduces the network traffic load due to VM migration by 85%, 89%, 76%, and 48%, and the total migration time by 66%, 71%, 59% and 40%, for Banking, Ecommerce, Support, and Specjbb, respectively. Unlike the Win7-64 test VM, introspection does not benefit the remaining iterations much even when it produces a significant benefit in the first iteration for all four test workloads running on the Debian-32 VM. In addition, the reduction percentage in total migration time is not as significant as the reduction percentage in memory state transfer, and the ratio between these reduction percentages is smaller in the Debian-32 VM than in the Win7-64 VM. For example, for the Banking workload running on the Debian-32 VM, the network traffic reduction percentage due to introspection is 85% but the migration time reduction percentage due to introspection is only 66%; for the E-learning workload running on the Win7-64 VM, the network traffic reduction percentage due to introspection is 44% but the migration time reduction percentage due to introspection is only 38%. The reduction percentage ratio is 0.77 (66%/85%) for the Debian-32 VM and 0.86 (38%/44%) for the Win7-64 VM.

In the current implementation, the migration module first asks the hypervisor to map all pages in the migrated VM and then queries the hypervisor for each page's type information. This mapping and query step incurs a fixed overhead, which accounts for the discrepancy between the reduction in amount of memory state transfer and the reduction in total migration time. More specifically, this fixed overhead becomes relatively more significant when the total migration time becomes smaller. Consequently, when the performance benefit of $IntroMigrate$ increases, the reduction in the total migration time increases, the relative importance of this fixed overhead increases and finally the discrepancy also increases. Because the performance benefit of $IntroMigrate$ is more pronounced in the Debian-32 VM than in the Win7-64 VM, the discrepancy between network traffic volume reduction and total migration time reduction is therefore larger in the Debian-32 VM than in the Win7-64 VM.

7. Conclusion

Memory virtualization provides many opportunities for performance optimizations, and these optimizations are more effective when more memory usage knowledge about guest VMs is available. In this research, we propose using VM introspection to extract a key guest kernel data structure, free memory pool, and apply this free memory pool information to improve the efficiency of memory de-duplication and memory state migration. To make VMI possible for multiple types and versions of guest kernels, we develop a *bootstrapping VMI* technique that reduces kernel version-specific hard-coding by programmatically leveraging as much publicly available information about a guest kernel as possible, including its source code, and demonstrate its effectiveness by applying it to multiple versions of Linux and Windows OS to successfully extract their free memory pool information without manual intervention. By leveraging free memory pool information, we show that the amount of memory de-duplicated *and* the de-duplication speed in memory de-duplication are improved significantly, and the memory state migration time during VM migration is substantially reduced. In summary, this research makes the following three research contributions:

- Development of a novel bootstrapping VMI technique that could identify the free memory pool of multiple versions of Windows and Linux OS with minimal kernel version-specific hard-coding,
- Leveraging free memory pool information to improve the speed and effectiveness of memory de-duplication, and
- Leveraging free memory pool information in reducing the network traffic load associated with memory state migration.

References

[1] Mac developer library, technical note tn2118, kernel core dumps. http://developer.apple.com/library/mac/#technotes /tn2004/tn2118.html.

[2] Amazon ec2. amazon's web service for virtual machine privision. http://aws.amazon.com/ec2/.

[3] Crash, linux crash dump analysis tool. http://people.redhat.com/anderson/.

[4] Microsoft debug interface access sdk. http://msdn.microsoft.com/en-us/library/x93ctkx8.aspx.

[5] Specjbb2005. http://www.spec.org/jbb2005/, .

[6] Specweb2009. http://www.spec.org/web2009/, .

[7] Sysmark2007. http://www.bapco.com/products/ sysmark2007preview/.

[8] Microsoft portable executable and common object file format specification. *ReVision*, page 97, 2010.

[9] A. Arcangeli, I. Eidus, and C. Wright. *Increasing memory density by using KSM*, pages 19–28. Linux Symposium, 2009.

[10] R. Bryant, A. Tumanov, O. Irzak, A. Scannell, K. Joshi, M. Hiltunen, A. Lagar-Cavilla, and E. de Lara. Kaleidoscope: cloud micro-elasticity via vm state coloring. EuroSys '11, 2011.

[11] C. Clark, K. Fraser, S. H, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live Migration of Virtual Machines. In *NSDI*, 2005.

[12] B. Dolan-Gavitt, T. Leek, M. Zhivich, J. Giffin, and W. Lee. Virtuoso: Narrowing the semantic gap in virtual machine introspection. SP '11, pages 297–312, 2011.

[13] T. Garfinkel and M. Rosenblum. A virtual machine introspection based architecture for intrusion detection. In *NDSS*, 2003.

[14] 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. OSDI '08, 2008.

[15] D. Magenheimer. *Transcendent Memory on Xen*, page 3. XenSummit, February 2009.

[16] D. G. Murray, S. H, and M. A. Fetterman. Satori: Enlightened page sharing. ATEC '09, 2009.

[17] B. D. Payne, M. D. P. D. A. Carbone, and W. Lee. Secure and flexible monitoring of virtual machines. *ACSAC 2007*, 2007.

[18] M. Russinovich, D. A. Solomon, and A. Ionescu. *Microsoft Windows Internals: Including Windows Server 2008 and Windows Vista, 5th Edition*. Microsoft Press, Redmond, WA, USA, 2009. ISBN 0735625301.

[19] M. Russinovich, D. A. Solomon, and A. Ionescu. *Windows Internals: Covering Windows Server 2008 R2 and Windows 7*. Microsoft Press, Redmond, WA, USA, 2011. ISBN 0735648735.

[20] M. E. Russinovich and D. A. Solomon. *Microsoft Windows Internals, 4th Edition: Microsoft Windows Server 2003, Windows XP, and Windows 2000*. Microsoft Press, Redmond, WA, USA, 2004. ISBN 0735619174.

[21] M. Schwidefsky, R. Mansell, D. Osisek, H. Franke, H. Raj, and J. H. Choi. Collaborative memory management in hosted linux environments. In *OLS06*, pages 313–331, 2006.

[22] C. A. Waldspurger. Memory resource management in vmware esx server. *SIGOPS Oper. Syst. Rev.*, 36:181–194, 2002.