Karlsruher Institut für Technologie

On Statistical Properties of Duplicate Memory Pages

Diplomarbeit
von

cand. inform. Thorsten Gröninger

an der Fakultät für Informatik

Erstgutachter:	Prof. Dr. Frank Bellosa
Zweitgutachter:	Prof. Dr. Hartmut Prautzsch
Betreuender Mitarbeiter:	Dipl.-Inform. Marc Rittinghaus
	Dipl.-Inform. Konrad Miller

Bearbeitungszeit: 1. Mai 2013 – 31. Oktober 2013

Abstract

In this work, we investigate the possibility to make memory deduplication scanners more efficient. Modern memory scanners equipped with hinting mechanisms merge large amounts of duplicate memory pages originating from disk, but still lack to harvest other replicas equally fast. We analyzed the properties of this remaining sharing potential and aim to decrease the amount of scanned pages by directly focusing memory scanners to stable page content. Stability is necessary to share content, or otherwise the sharing is instantly broken. With a metric to exclude unstable pages, it is possible to speed up merging.

We acquired memory modifications and semantic information with a full-system simulation to analyze sharing opportunities, memory access frequencies, and access patterns which lead to stable pages. We implemented a toolchain that allows to gather such information quickly and scalably. Our evaluation shows that up to 89% of all pages are stable and can be shared with other VMs executing the identical file benchmark. Furthermore, a heuristic for CPU or I/O bound workloads can only exist for a small sub-set of examined workloads, e.g., kernel builds. General page state prediction seems impossible.

Our findings show that memory write frequencies correlate with page stability, even in otherwise unpredictable workloads. About 78% of all pages experience a low access frequency before they stabilize. A memory scanner should therefore prioritize pages that show a low write access frequency. A reasonable threshold appears to be about 4 accesses within a window of 1.5 seconds. Pages with high memory access frequencies such as device associated page frames can be excluded permanently from scans, if their overall busy time exceeds 15 seconds. We further conclude that a scanner should focus on pages leaving the write working set instead of linear scanning all pages. These pages (on average about 1,800 pages per 480 ms) are guaranteed to have been recently modified, but are not currently written and are thus candidates for further examination by a scanner.

I hereby declare that this thesis is my own original work which I created without illegitimate help by others, that I have not used any other sources or resources than the ones indicated and that due acknowledgment is given where reference is made to the work of others.

Thorsten Gröninger
Karlsruhe, October 31th 2013

Deutsche Zusammenfassung

Heutige Computersysteme werden mit immer mehr Arbeitsspeicher ausgestattet, allerdings wächst gleichzeitig auch der Speicherbedarf durch Virtualisierung, In-Memory Datenbanken und Cloud Computing. Obwohl Speicher erschwinglich erscheint, bleibt er (zu einem gewissen Grad) eine begrenzte Ressource. Insbesondere Virtualisierung erzeugt redundante Daten; große Teile des physischen Speichers enthalten identische Inhalte und könnten eigentlich dedupliziert werden. Allerdings sind die heute verfügbaren Deduplikationslösungen für Arbeitsspeicher nur eingeschränkt nutzbar. Zum einen benötigen sie speziell angepasste Betriebssysteme, Virtualisierungslösungen und Programme. Zum anderen verbrauchen sie sehr viel Rechenleistung um redundante Speicherseiten zu finden und zusammenzulegen, dieser Vorgang kann durch verschiedene Ansätze verbessert werden.

XLH (Cross Layer Hints) [35] benutzt den in Linux verfügbaren Deduplikationsmechanismus und kombiniert ihn mit Informationen aus dem virtuellen Dateisystem um schneller den Speicherverbrauch einer virtuellen Maschine zu reduzieren. Verschiedenste empirische Untersuchungen [9, 35, 36] haben gezeigt wie erfolgreich Speicherseiten, deren Inhalt von der Festplatte geladen wurden, dedupliziert werden können. Allerdings müssen die anderen Speicherseiten, z.B. anonymer Gastspeicher, immer noch linear durchsucht werden. Große Teile des theoretisch teilbaren Speichers bleiben unberücksichtigt oder werden nur sehr langsam zusammengelegt, dies sind, abhängig von den jeweils ausgeführten Programmen, bis zu 50% aller theoretisch teilbaren Speicherseiten.

Diese Arbeit beschäftigt sich mit diesem ungenutzten Deduplikationspotential, dessen Eigenschaften und Verbesserungsmöglichkeiten für Memory Scanner. Wir untersuchen das Potential von duplizierten Speicherseiten, ihre Stabilität und Speicherzugriffsmuster die zu (in-)stabilen Seiten führen, um jene zeitweise oder dauerhaft von einem Scan auszuschließen.

Um eine tiefergehende Untersuchung dieser Speicherseiten durchzuführen, wurden die entsprechenden Daten, z.B. alle Speicherzugriffe, aufgezeichnet. Hierfür wurde ein Systemsimulator verwendet, um eine ungewollte Beeinflussung der ausgeführten Benchmarks auszuschließen. Dieser simuliert Hardware auf der unmodifizierte Betriebssysteme ausgeführt werden können, erlaubt somit auch alle

Speicherzugriffe transparent zu unterbrechen und aufzuzeichnen. Für diese Arbeit wurde eine Version des freien, quelloffenen Simulators QEMU verwendet, der um die benötigten Schnittstellen erweitert wurde. Da eine Simulation um mindestens zwei Größenordnungen langsamer läuft als ein reales System, haben wir die Aufzeichnung der Daten von der Analyse getrennt. Die Speicherzugriffe und die semantischen Informationen innerhalb des Betriebssystems werden erfasst, über einen Speicherserver komprimiert und für die spätere Verwendung abgelegt. Unsere Analysen nutzen die aufgezeichneten Daten und rekonstruieren ein Speicherabbild für die eigentliche Duplikatsanalyse. Diese wird wiederum mit semantischen Informationen aus dem untersuchten Linuxkern ergänzt. Dieses Vorgehen ermöglicht eine verfälschungsfreie Untersuchung des noch nicht genutzten Deduplikationspotentials, auch unter Einbeziehung der eigentlichen Verwendung einer Speicherseite innerhalb des Betriebssystems.

Unsere Untersuchungen bestätigen das Potential, das in Speicherseiten steckt, die von der Festplatte eingelesen wurden. Gleichzeitig zeigen sie aber auch die Abhängigkeit von den ausgeführten Programmen, z.B. können Datenbankanwendungen, die häufig ihre Daten modifizieren und zurück auf den Hintergrundspeicher schreiben, nur im geringeren Umfang davon profitieren. Das verbleibende Deduplikationspotential, das nicht vom Hintergrundspeicher stammt, ist noch wesentlich instabiler und variiert stark bei unterschiedlichen Anwendungen, z.B. verursachen Simulationen, wie sie in der Physik Anwendung finden, große Mengen an identischen Speicherseiten. Allerdings verändern sich diese derartig schnell, dass selbst ein spezialisierter Speicherscanner nicht in der Lage ist diese Seiten zu vereinen.

Ein Speicherscanner sollte sich allerdings die Tatsache zu Nutze machen, dass Speicherseiten unterschiedlich oft beschrieben werden und cache-affine Programme nur überschaubar viele Speicherseiten in einem Zeitinterval ändern. Die meisten von uns untersuchten Anwendungen ändern im Mittel ungefähr 1.850 Seiten (7,25 MiB) innerhalb einer halben Sekunde. Dieser modifizierte Arbeitsbereich kann von einem Scanner leichter bewältigt werden, als den gesamten Arbeitsspeicher regelmäßig eines aufwendigen Vergleichs zu unterziehen.

Mit der gleichen Methode wie sich das Working Set ermitteln lässt, können auch Speicherzugriffsfrequenzen ermittelt werden. Unsere Untersuchung zeigt, dass Seiten die stabil werden, also ihren Inhalt für ein gewisses Zeitfenster nicht ändern, in über 80% aller Fälle bereits vorher eine sehr niedrige Zugriffsfrequenz hatten. Leider reicht es nicht aus die Stabilität einer Speicherseite zu ermitteln, um direkt auf einen Kandidaten zur Deduplizierung zu schließen, dies gilt nur für Dateien die von einem identischen Festplattenabbild geladen wurden. Allerdings erweist sich Stabilität als sehr wichtig, denn sie stellt sicher, dass eine Seite nicht direkt nach dem Zusammenlegen wieder kopiert werden muss und so unnötige Rechenleistung vergeudet wird.

Im Laufe unserer Untersuchungen hat sich des Weiteren gezeigt, dass ein Memory Scanner, ungeeignete Speicherbereiche, die für Kernelstacks oder Geräte verwendet werden, dauerhaft von einem Scan ausschließen sollte, da diese Seiten sich häufig ändern und keine ausreichende Stabilität zeigen, um sie gegebenenfalls zu deduplizieren.

Wir haben festgestellt, dass die einzige nutzbare Eigenschaft von redundaten Speicherseiten ihre Stabilität ist. Allerdings lässt sich trotzdem nicht allgemein sagen, ob sich stabile Speicherseiten in Zukunft zusammenlegen lassen.

Contents

Chapter 1

Introduction

The demand for main memory is increasing every year. In-memory databases and virtualization push the demand for memory even further. As virtualization has become widely used in cloud computing, memory consumption and data redundancies have reached a high level, for example, up to 79% [11] of data is redundant in such scenarios.

Although main memory is not a rare resource in most server systems, it still can be used for more *virtual machines* (VMs) than holding replicated data. The only justifiable redundancies are found on different *non-uniform memory access* (NUMA) nodes, where memory access comes at different costs and a local copy increases performance. In other cases, memory footprints of VMs and applications can be reduced by using memory deduplication techniques.

There are two different kinds of deduplication mechanisms. Firstly, systems that try to prevent the replication of data by proactively establishing shared pages instead of copying content. Widely used techniques are shared libraries and forking with *copy-on-write* (COW), or in a virtualized environment: VM forking [27]. Secondly, deduplication systems actively search all pages, identify replicated content and merge them to a single copy. This method is called memory scanning.

Memory scanning reduces the memory footprint of applications and VMs by detecting identical memory content on page granularity and remapping them to a single copy-on-write page. A recent empirical study shows that the amount of redundant data ranges from 11% up to 79% [11]. If this sharing potential can be harvested, the reduced memory footprint allows more VM instances to run in parallel. Furthermore, the CPU cache utilization and its hit ratio increases [41] and thus allows faster program execution.

Although, the sharing potential is already harvested by different deduplication approaches, there is still room for improvement. Even the most promising techniques, for example, Cross Layer Hints (XLH), still fail to fetch about 50%

of theoretically shareable content in some workloads [35]. XLH uses hints from the I/O layer of the hosting OS and focuses its memory scan on pages originating from file operations. Although, these hints speed up deduplication for page cache pages in different VMs, other sharing opportunities must still be collected through periodical linear memory scanning.

Objectives

This thesis focuses on properties of unharvested sharing potential and tries to find new hinting sources to improve scanner based memory deduplication. We examine options to focus scanners on promising memory regions and determine memory pages that can be excluded from the scan.

The effectiveness and efficiency of a scanner depends on the stability of scanned and merged pages, we clarify the relations between semantics and stability.

Contributions

To analyze sharing opportunities and to identify new hinting sources, we collect memory modifications and semantic information from within a guest operating system (OS). We regard memory operations, trace OS page frame allocations, and the OS system state without influencing the guest OS by using a full-system simulator for our experiments. A simulator does not only hide memory inspection overhead from the guest OS, it is also the only feasible method to collect all memory modifications. Therefore, we augmented a simulator to allow flexible and fast recording of memory accesses and semantic information, store them efficiently, and replay memory modifications to reconstruct memory content for every point in time. We were able to perform false positive free sharing analyses, memory access frequency and pattern measurement, and enriched these data with semantic information.

Our evaluation is based on data acquired with the fast, open-source simulator QEMU and analyzed with an adapted framework of a previous research project [41]. To cover a wide-range of different use-case scenarios, we evaluated I/O-, CPU- and memory bound workloads, and a mixture of realistic database and filesystem benchmarks.

Our findings show that memory write frequencies correlate with page stability, even in otherwise unpredictable workloads. About 78% of all pages experience a low access frequency before they stabilize. A memory scanner should therefore prioritize pages that show a low write access frequency. A reasonable threshold appears to be about 4 accesses within a window of 1.5 seconds. Pages with high memory access frequencies such as device associated page frames can be excluded

permanently from scans, if their overall busy time exceeds 15 seconds. We further conclude that a scanner should focus on pages leaving the write working set instead of linear scanning all pages. These pages (on average about 1,800 pages per 480 ms) are guaranteed to have been recently modified, but are not currently written and are thus candidates for further examination by a scanner. Memory access patterns, however, do not show any predictability for sharing opportunities or at least stability. In contrary they mispredict the future development of pages and are often beaten by normal distributed random numbers. Fine-grained predictions, based on a hidden Markov model, which show good prediction rates for file deduplication, suggest that memory stability can only be predicted for each workload individually.

Organization

The thesis is structured as follows. In Chapter 2 we describe memory deduplication techniques and recent memory deduplication systems, and introduce full-system simulation. In Chapter 3 we discuss our analysis objectives and define the data required for these analyses. Our design is presented in Chapter 4 followed by the description of our implementation in Chapter 5. The results of our evaluation can be found in Chapter 6 followed by our conclusion and future work in Chapter 7.

Chapter 2

Background and Related Work

Memory is a valuable resource and the demand for memory is increasing as more data is continuously produced and stored. Virtualization in conjunction with cloud computing, in-memory databases, and big data increase the demand for memory. Deduplication of files is a common technique for efficient background storage devices [32]. Similar techniques can be applied to main memory as well.

This chapter presents an overview of memory deduplication techniques, from early memory sharing approaches (*forking*), to the latest developments and improvements for VMs (*Cross Layer Hints*), and native applications (*Page Combining*). Additionally, this chapter gives an introduction to full-system simulation, its fundamental modes of operation (*binary translation*), and different simulation modes (*functional* and *micro-architectural*). It provides an overview of recent simulators, e.g., *Quick Emulator* (QEMU).

2.1 Memory Deduplication Techniques

Many techniques have been developed to reduce memory footprints of applications, *virtual machines* (VMs), or complete systems.

The overall sharing potential varies with the *operating system* (OS) and the executed workloads. It further depends on the mixture of workloads executed in different VMs. These redundancies vary from 11% to 79% [4, 11, 26] of all pages, considered for sharing.

Figure 2.1 presents an overview and relations between the different approaches to reduce memory footprints. We differentiate between two categories of deduplication. In the first category, a VM or application is aware of the deduplication, and must be modified, and therefore *cooperate*, to benefit from deduplication, e.g., a paravirtualized OS. In the second category, deduplication can be applied without any cooperation and therefore tasks and VMs can benefit.

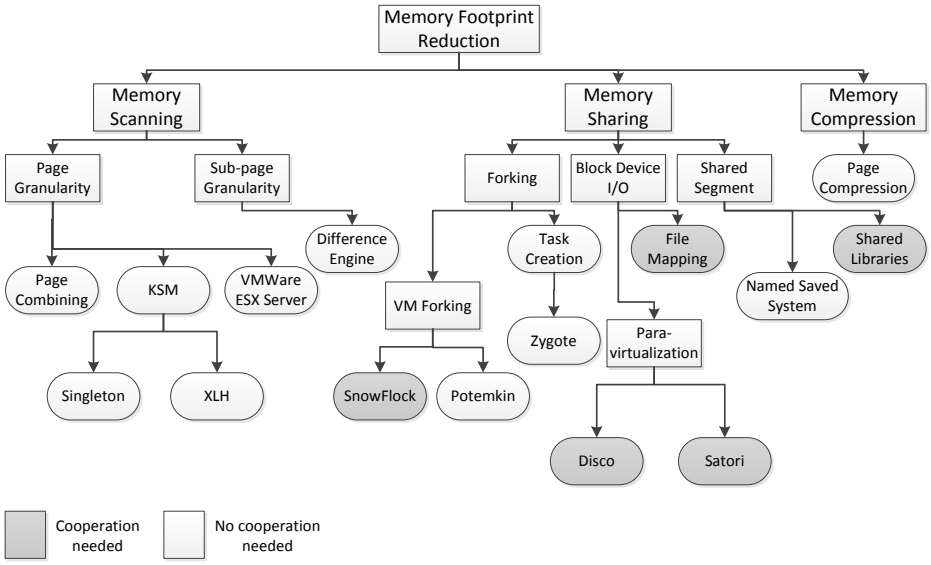

Figure 2.1: Overview and relations of memory footprint reduction techniques.

Memory deduplication aims to reduce the amount of consumed physical memory in two ways. Firstly, memory sharing, such system prevents the replication of data to different page frames proactively, i.e., it establishes shared pages instead of copying the content. Secondly, memory scanning, already replicated data is identified and merged re-actively to a single copy. Both methods reduce the memory redundancy. The amount of computational power of both approaches diverge: Where proactive mechanisms typically consume less CPU resources as re-active ones.

2.1.1 Conventional Approaches

Virtual memory was invented in the 1950s and became available for commercial computers during the late 1970s. The virtualization of address spaces isolates programs from each other, allows paging content to disk, to increase the amount of available memory, and simplifies programming [29]. However, it also consumes more physical memory, since parts of programs have to be kept in redundant copies, compared to a single address space. As physical memory was much more limited than today, the first ideas of deduplication were invented, e.g., forking with copy-on-write to share memory content.

Traditional approaches contain two categories of proactive sharing. Firstly, *semantic sharing*, which considers libraries and files and secondly, *forking*.

Semantic sharing An OS can benefit from shared libraries, as an application only must be linked dynamically. The operating system's binary loader can map libraries into every address space, but must only retain a single copy in physical memory. Sharing is possible at per library granularity. Slightly different versions require an extra copy of all library associated pages, although only few pages might have changed. (Although different instances of such applications can still share their executable image.)

The idea of sharing libraries can be extended to all files. Commodity OSes provide mechanisms, e.g., file caches, for this purpose, but as for shared libraries, the application must explicitly use the file mapping mechanisms. Applications cannot rely on standard file APIs, or the standard C library, which make private copies into supplied buffers, rather than mapping the file's content into its address space. Afterwards two identical copies occupy memory, one in the address space and one in the file cache of the OS. These issues and the impossibility of sharing memory not backed by files (*anonymous memory*) with child processes led to the idea of *forking with copy-on-write semantic* [45].

Forking The process creation on POSIX-compatible operating systems invokes the fork system-call, which makes a shallow copy of the calling process' *virtual address space* (VAS). After this call, two identical virtual address spaces exist, sharing all content [45]. Both page tables point to the same physical memory, but to retain address space isolation all pages are marked as *copy-on-write* (COW). The OS creates a private copy of COW pages, if a write access occurs. That allows tasks to work without modification. However, as long as a identical task continues to execute inside the VAS, large parts of memory remain shared and only few pages, typically on the stack and heap, are copied.

2.1.2 Initial Sharing Across Domains

The traditional deduplication techniques rely on relations between objects, such as libraries and files. This information is not available from outside hardware-accelerated VMs without guest modifications. A VM forms an *isolation domain* with an independent system state, which separates the guests from its host. Therefore, the *virtual machine monitor* (VMM) lacks semantic information about memory allocations or open files within the guest operating system. It might not even be able to distinguish files on the used *virtual disk image* (VDI). As a result the traditional techniques cannot be applied to non-paravirtualized VMs, and new content deduplication schemes are necessary to deal with this situation.

VM Cloning VM cloning was first implemented in Potemkin [51]. It leverages the same concept for VMs as the traditional fork for standard processes. Instead of allocating and booting a new VM. Potemkin creates an identical copy of an already running – *pre-warmed* – VM. This way, the virtual machines can share large parts of their allocated memory, e.g., the operating system kernel image. VM cloning, as done by Potemkin, allows a fast instantiation of VMs and over-committing of physical memory. Although the primary intention was malware analysis in *honey pot* server farms, it has influenced cloud computing and other hypervisors.

Snowflock [27] extends this approach and includes a hypercall interface so every VM can signal the need of more resources or a further instance to cope with, e.g., incoming connections in a virtual server environment. The hypervisor can create a shallow copy of the VM, sharing most of their associated pages and keeping the memory consumption low.

The most prominent use of VM forking is the Zygote process of Android [8]. Since Android needs to run on energy and memory limited devices, it has to preserve resources whenever possible. To prevent content duplication, the DalvikVM, the Java runtime environment of Android, is hosted inside the *Zygote process* and forked on demand. This pre-warmed VM has already produced its common data structures and mapped all necessary libraries and resource files into its virtual address space. Thus, a fork enables a newly started application to share nearly all pages and the contained resources with its parent. Only few and typically fast changing pages are private to each process; in the case of the DalvikVM, the runtime generated code, as well as, heap and stack pages diverge.

Named Saved System (NSS) A different approach available on IBM's System Z are *Named Saved Systems* (NSS) [22]. NSS are similar to shared libraries. A NSS contains an OS binary image and the hypervisor maps it during VM initialization into the guest memory. These segments are shared across a large amount of VMs, with only one backing copy in memory. A *Discontinuous Saved Segment* (DCSS) [21] can also contain more shareable memory regions, e.g., parts of an already booted VM, which is instantiated and reinitialized by the z/VM hypervisor, allowing cloned, pre-warmed VMs with content sharing.

In-place Execution A slightly different concept available for Linux on System Z, is *In-place execution* [21]. Originally designed for embedded systems with limited RAM, it allows different VMs to load executable files from background storage without copying the content into VM private memory, instead it can directly execute them from a single copy in memory. Thus large and common binaries can be easily shared between different VMs without consuming additional memory.

A drawback is that the guest OS must be aware of this technique and requires a modified filesystem driver to allow direct execution.

2.1.3 Paravirtualized Deduplication

A major drawback of VM forking – and forking in general – is, that once a COW page's content has changed, it is never shared again, although the contents might be redundant in memory or change back in the future. In paravirtualized environments the guest OS is aware of running inside a VM and communicates directly with its hypervisor. The hypervisor can therefore easily gain information about guest memory operations and can, for instance, intercept memory copy operations and instead establish respective page mappings to prevent memory duplication. It can further provide host to guest file cache sharing.

DISCO Bugnion et al. extended the DISCO hypervisor to serve as the basis for *transparent page sharing* [9]. This technique allows, in conjunction with a copy-on-write virtual disk, to share the content of the host's and guest's page caches. Copying the file content into each virtual machine instance is unnecessary; instead DISCO just maps the corresponding pages. All instances share a single copy of the disk content on a block granularity. As long as the same *virtual disk image* (VDI) is used and no write access takes place, no further copy operation is required. If file content is modified a private copy is created. Although this approach is limited to paravirtualized systems with special COW disk images and only available for few operating systems, it still influenced further memory deduplication techniques [52].

Satori Satori [36] extents the Xen hypervisor with similar capabilities as DISCO, but does not rely on a single shared disk. In a common usage scenario, every VM has its private VDI; therefore sharing content based on the same logical address is impossible. To overcome this limitation Satori allows sharing based on a page's content, rather than on disk block addressing. Every page gathered from backing store is hashed and inserted into a look-up structure. A request for identical content, from a different VM, causes a look-up and verification. On success the matching page frames are mapped into the requesting VM directly. It ensures that read-only data is present only once in system memory and not scattered in various copies across different VMs. A huge limitation of Satori and DISCO is that they cannot track changes in *anonymous memory* such as heaps and can only utilize data emerging from I/O requests in a paravirtualized environment.

2.1.4 Memory Scanning

Memory scanners detect shareable content by periodically scanning memory pages, typically uncorrelated to a certain system event. Furthermore, memory scanners need no cooperation of the guest OS running inside a VM. Memory scanning can therefore be applied to OSes were paravirtualization is unavailable, or even impossible, due to lack of source code or copyright issues. Furthermore, it can also be applied to native applications, even to applications which do not benefit from traditional sharing techniques, i.e., forking or shared libraries. Memory scanners primarily focus on *anonymous memory*—pages allocated for heaps and stacks.

Although memory scanners differ in their heuristics, scanning intervals, merging granularity, and look-up structures for identical page identification, their common goal is to minimize memory usage of applications and VMs. Once a scanner has identified a pair of duplicate pages, it transparently replaces them with a single copy-on-write protected page, freeing the redundant page copies.

VMWare ESX Server The hypervisor VMWare ESX Server includes one of the first memory scanners [52]. Its primary aim is to reduce the memory consumption of VMs and allow more VM instances to run on the same system—for server consolidation. It does not require any adaptations of the guest OSes. The scanner works on page granularity and uses the hash of a page's content as an index into a *hash map*. This look-up structure reduces identification and verification complexity. The scanner is invoked periodically, i.e., every 20 minutes, or just before the hypervisor has to swap-out pages to comply with memory demands of VMs. This solution is dedicated to VMs only.

Kernel SamePage Merging (KSM) The Linux kernel has been extended with a memory scanner, which is upstream since Linux kernel version 2.6.32. It is called *Kernel Samepage Merging* (KSM) [3]. Its main purpose is to re-actively share *anonymous memory* across different virtual address spaces. Sharing is possible between regular tasks and VMs – hosted on the *Kernel Virtual Machine* (KVM). That is possible since every KVM-hosted machine allocates only *anonymous memory* to provide guest physical memory. KSM considers pages inside previously marked virtual memory regions and ignores file-backed pages, since Linux does not yet provide a method to merge them.

KSM runs in a single kernel thread that wakes up periodically to scan a configured amount of pages, ignoring current CPU utilization and resource limitations. It utilizes red-black trees to identify possible *shareable* pages. For this purpose, KSM maintains an *unstable tree*, containing all scanned pages, and a *stable tree*, which includes all pages already shared. The index into these trees is formed by the content of pages. A major drawback of this indexing scheme is that changes to

a page and therefore to the index are not tracked. The unstable tree can degenerate drastically since it is only rebuilt after a full scan of all advised pages [3]. The chances of finding mergeable pages declines rapidly [35] and ignores many shareable pages. As in VMWare ESX Server and other scanners, Linux uses copy-on-write to protect merged pages from modifications. The efficiency of KSM depends heavily on the workloads executed. CPU- and memory bound workloads are least likely to benefit from KSM's scanning mechanism, whereas an I/O bound job, using large amounts of the guest's page cache, can benefit to a greater extent [11].

Page Combining Recent versions of Windows – namely Windows 8 and Windows 2012 Server – also include a memory deduplication technique for *anonymous memory*. This mechanism is called *Page Combining* [19]. It aims to reduce the memory consumption of user land programs. In contrast to KSM, Page Combining operates on *page frames* and not on *virtual address space regions*, thus, reducing the quantity of scanned content and the overhead caused by page table walks. It does not provide a mechanism to explicitly include or exclude page frames. Instead a default filter is integrated in the Windows kernel. It decides whether a frame is compatible for merging or not. In general, it excludes all frames dedicated to device drivers, working sets, free or zeroed memory lists, and pages already shared by file mappings or shared libraries. Another difference to KSM is its runtime behavior. Whereas KSM wakes up periodically and scans a certain amount of pages, Page Combining, once activated, wakes up in idle periods and scans the whole physical memory in a burst, as long as no other activity interrupts the scan. The background service responsible to host the scan thread wakes up and scans each NUMA node of the system. It acquires the necessary locks, to inhibit memory allocations and scans the *page frame database* of each node. Checksums for every suitable page frames are calculated and sorted to speed-up duplicate detection. Afterwards all identical page frames are merged to a single copy. The kernel maintains a simple hash table with binary trees to identify merging candidates in future passes more quickly. However, as noted earlier, Windows excludes driver associated page frames and focuses on user land programs. Due to this limitation, VMs cannot benefit from shareable pages between different VMs [46]. Instead, Page Combining is intended to run *within* the VMs, opposed to KSM.

Difference Engine The memory scanners, work on *page granularity*. Difference Engine [18] takes a different approach and identifies identical content on a sub-page granularity. For each found pattern, it creates shareable patched pages. That decreases the memory footprint of VMs beyond the other scanners, but it comes at a cost. Difference Engine has to ensure that the correct page content is

present on each memory access, thus, all patched pages are mapped inaccessible. Therefore, it can only be applied to pages, neither read nor written. Otherwise, page restoration overhead might easily exceed its benefits.

2.1.5 Hinting

Pure memory scanning consumes lot of computational power and still it takes long periods of time – up to minutes – to identify identical memory content [35]. Thus, a hinting mechanism, utilizing semantic information, e.g., disk accesses, can focus a scanner to promising pages, avoiding exhaustive scans.

Cross Layer Hints (XLH) Previous research suggested a strong correlation between sharing opportunities and *named memory* [36]. To overcome the limitations of memory scanners and paravirtualized guests, *Cross Layer Hints* (XLH) [35] combines an improved scanning mechanism of KSM, with hints from the host's I/O layer. Regular scans take a long time – up to five minutes for small amounts of memory (1 GiB) [35]. This time penalty – for only one pass – prohibits KSM to harvest short living (between 10 and 30 seconds) identical memory pages. Furthermore, it cannot share them, because the *unstable tree* degenerates fast and is seldomly rebuilt. XLH introduces two improvements to KSM. Firstly, it prevents the degeneration of the *unstable tree* by tracking page content changes. Secondly, every time an I/O operation takes place, the virtual file-system layer issues a hint to XLH. These hints point to potentially shareable pages, and are therefore considered before any other page in the linear scan. On average, such I/O related pages are merged five times faster than with the pure improved scanning mechanism.

Singleton A different optimization of KSM is taken by Singleton. Singleton utilizes the merging capabilities of KSM [44] and also uses hints from the I/O layer of the hosting hypervisor. In contrast to XLH, it uses the hints to search KSM's stable tree for a match and on success, it tries to remove these identical pages from the host's page cache. It only retains a single content copy for all guest page caches, in contrast to XLH, where an additional copy is still available in the host's page cache. It, therefore, reduces overall memory consumption further than the I/O based hinting scheme of XLH can. It also avoids modifying large parts of the Linux memory subsystem, which otherwise must be adapted to allow sharing named memory pages with anonymous pages and vice versa.

2.1.6 Conclusion

Although there are many different approaches, which reduce the memory footprint of applications and VMs, all of these techniques are far from perfection or concentrate their afford only on parts of the deduplication potential.

Memory sharing proactively avoids content duplication, but is either limited to named memory such as DISCO and Satori, or cannot reunify duplicated content, for example, VM forking or Saved Named Systems. In contrast, memory scanning re-actively merges identical memory content, but consumes much computational power to harvest shareable content.

For instance, even a reasonable configured memory scanner can only fetch less than 50% of anonymous guest pages [35]. There is no system yet, which harvests nearly all shareable pages with low overhead. Furthermore, to avoid exhaustive scanning, a guest OS must still be instrumented to re-actively share anonymous pages, which is in general not a satisfying solution.

2.2 Duplicate Memory Content

As most applications and VMs produce redundant data on a page, or sub-page granularity, previous work tries to reduce memory footprints by merging content into a single copy [3, 18, 35]. This identical and redundant content is called a *sharing opportunity*. Deduplication tries to harvest these opportunities to decrease the memory consumption. There are two different types of sharing opportunities, an *intra-domain* sharing opportunity – also called *self-sharing* – and an *inter-domain* sharing. They differ mainly in the pages considered for merging, as depicted in Figure 2.2.

Intra-domain In this case, only pages within the same domain, e.g., NUMA node or the same VM are considered for merging. These sharing opportunities are always available, whether a similar workload is executed or not. Intra-domain sharings are similar to compressed files, where only a single file is used to find compressible patterns, and all other files are ignored. In most cases these sharings are by a magnitude smaller than inter-domain sharings [26]. Their primary sources are redundancies within files and typical desktop workloads such as browsers, office applications, etc. [41].

Inter-domain In contrast to intra-domain sharings, all content of all accessible VMs is considered for matches. In a non-virtualized environment, inter- and intra-domain sharings are identical, since there is no additional domain. The *inter-domain* sharing heavily depends on the workloads residing inside the VMs. Har-

vesting inter-domain sharings must be done with care. Merging pages for instance across different nodes in a NUMA system, might slow down overall application performance. Furthermore, depending on the deduplication mechanism, security concepts such as strict domain isolation might be weakened [48]. Intra-domain sharings do not interfere with protected resources and remain in their own restriction boundary, whereas inter-domain sharings break this boundary. However, their sharing potential is higher and can reduce memory usage up to 79 % [41].

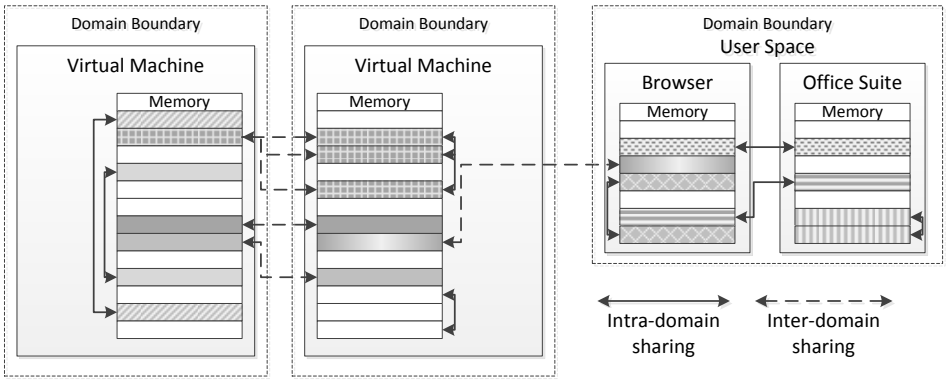

Figure 2.2: The figure shows two VMs (two semantic domains), and native applications, a domain of its own. Pages can be shared within the same domain (*intra-domain*) and across domain boundaries (*inter-domain*).

2.3 Full-System Simulation

Full-system simulation enables a thorough analysis of computer systems from coarse- to fine-grained detail. They enable the analysis of program behavior, memory operations, and other properties *without interrupting* the data flow, changing *memory content* or the *timing* of the guest system. In contrast to user land introspection tools, i.e., Valgrind [38] or Pin [30], full-system simulators provide a compatible system environment – a bare simulated hardware – for the guest OS. They further offer different levels of simulation detail, starting from pure *system-call emulation*, over *functional simulation*, to timing accurate *micro-architectural simulation*. There are two kinds of execution modes for this purpose: An *interpretive mode*, where each guest instruction is directly emulated via respective simulator routines and a *translation mode*, where, prior to execution, instructions

from the guest system are translated into a semantically identical instruction sequence of the host's *instruction set architecture* (ISA) [5]. The primary aim of the interpretive mode is typically a complex simulation of pipelines, CPU models and timing, whereas the translation mode favors speed and emulates only the result of an operation at a function level. Both approaches can be combined to achieve different levels of detail and increase execution speed.

2.3.1 Binary Translation

Every simulator has to decode and transform guest instructions into an operation, which can be executed on the host system. As illustrated in Figure 2.3, the primary step is to *disassemble* the guest code, and then feed this information into a *binary translator*. The binary translator will produce a semantically identical code sequence, which can be executed directly on the target platform. All produced code sequences alter only guest memory and the simulated system state [5].

Binary translation is the core element of most simulations. It reduces the bottleneck of conventional, interpretive emulation and speeds-up the execution of a simulated system [5]. Binary translation can be done in two flavors, *statically* or *dynamically* [40]. Static binary translation is done in advance, and no extra instruction decoding or compilation must be done during runtime. However, it has major drawbacks; it cannot translate all code paths correctly, since some information such as conditional jumps or privileged operations depending on the current CPU state are unknown in advance. Furthermore, *dynamic code generation*, often used by Just-In-Time compilers or *self-modifying code* cannot be used at all. Static code translation is thus not applicable to all programs and especially not feasible for whole operating systems. Full-system simulators thus typically execute guest code in an *interpretive mode* or switch to *dynamic translation*.

Dynamic binary translation (DBT) has been used in many system simulators and in early x86-compatible VMMs (VMWare Workstation, VirtualPC, etc.) [1] before hardware accelerated virtualization became available. It overcomes the drawbacks of static translation and enables the simulation to provide all features as present in the simulated ISA. The translation process, as depicted in Figure 2.4, is executed during the simulation and transforms the guest code on demand, following the current program flow of the guest. That allows the guest OS to run unmodified. However, it can be transparently extended with introspection functionalities. That is because, the resulting binary code, with modification and extensions, is invisible for the guest system.

The translator breaks the guest code down into *basic code blocks* [50]. A code block contains consecutive instructions till the next jump instruction. A jump terminates each basic block, since the jump's destination is unknown in advance and has to be evaluated after execution. This code partitioning allows to translate a

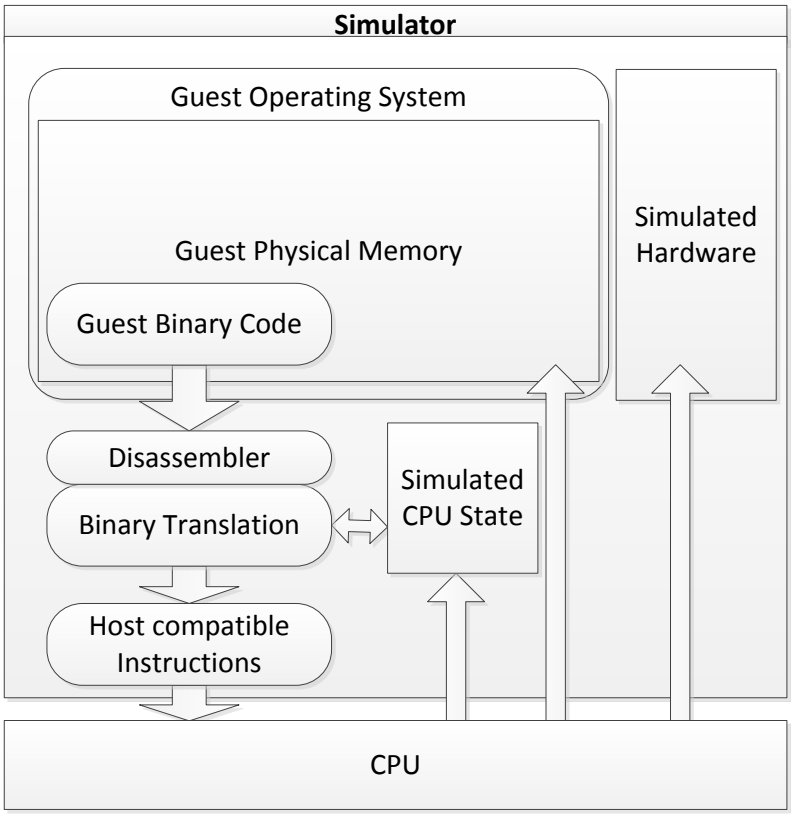

Figure 2.3: The general work-flow of a simulator using *binary translation*. The guest code is disassembled, translated, typically instrumented, and executed on the hosting platform. All memory accesses have been modified to point into the guest assigned memory regions.

complete block into its host equivalent. During the execution of the translated block, *program exceptions* or CPU state changes can occur, which require a *code block* to be modified and re-executed. For instance, if a *divide-by-zero exception* occurs, the following instructions cannot be executed without handling it. Therefore, the translator creates a new slightly different code block handling the exception and re-executes it. If a code block has been successfully executed, the translator can add it to its optional *translation cache* [40] to by-pass the complex block creation in the future. However, to support *self-modifying code*, already cached code blocks must be invalidated, if the guest code changes. Afterwards, the translation cycle starts again, at a new *Instruction Pointer* (IP).

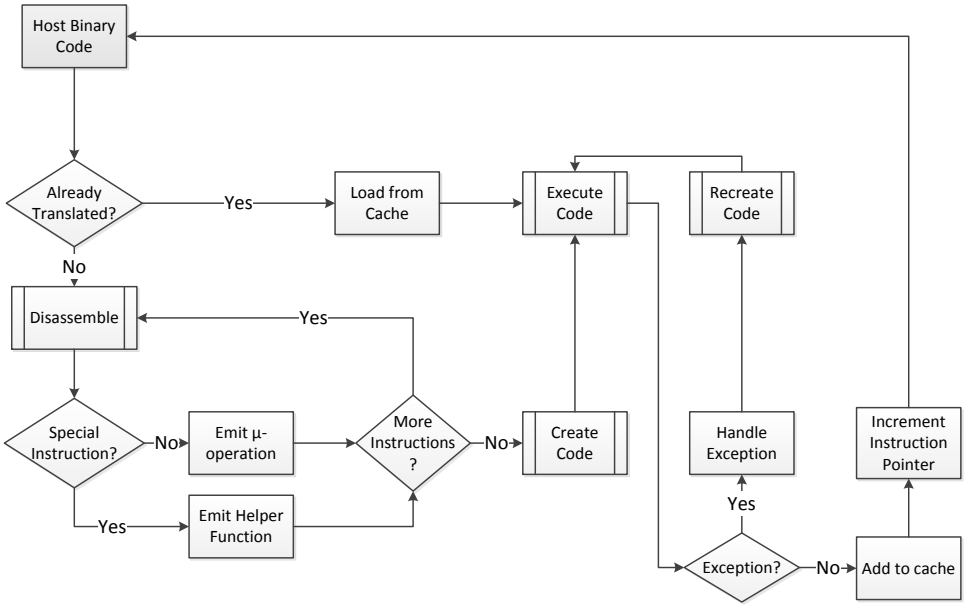

Figure 2.4: Dynamic binary translation and the typical work-flow of disassembling, translation, and execution to correspond to the guest ISA. The binary translator must recreate a *code block*, if an exception, e.g., a page fault has occurred. A successfully executed block is added to a cache to speed up re-execution.

In more detail, the *dynamic binary translation* for full-system simulators follows the following four basic steps.

Disassembling The guest code is disassembled, split up into its *instructions* and its *operands*. Some ISAs, e.g., ARM, are easier to disassemble than others, e.g., x86. The decoding of instructions can only be done as far as no conditional jump is encountered [40]. As jumps change the instruction pointer, all following instructions might become invalid and can therefore only be decoded after the execution of the jump. A problem of the x86 ISA is its *variable* instruction length. An instruction ranges from one up to 15 bytes, and are not aligned on a machine word boundary [23], which makes decoding even more complicated [1]. Once an instruction has been successfully decoded, it can either be directly passed to the *recompilation*, or it can form an *intermediate representation* (IR) [5]. An IR aims to match two different instruction set architectures. It typically consists of much simpler operations (load, store, arithmetic, branch instructions, etc.) – a sub-set of all supported instructions.

Recompilation and Instrumentation The disassembled code or its intermediate representation cannot run unmodified, since it still contains wrong, or incompatible memory addresses, privileged, or even unsupported instructions. They would alter the state of the host system, rather than the state of the simulation. Thus, addresses must be recalculated to map into the *guest's memory*, or at least comply with the simulated *memory management unit* (MMU). All privileged instructions have to be replaced with equal functionality, otherwise they cannot be executed at all. Furthermore, modifications to a simulated device state or the configuration of a simulated CPU would be unavailable, since typically these instructions can only be executed in a privileged execution mode. The simulator also needs to add exception handling code to cope with faulty memory accesses or to comply with the guest ISA's exception mechanisms. After these necessary modifications to the guest code, the simulator can further add *instrumentation* code, for system state *inspection*. That allows to add tracing functionality such as memory or instruction hooks. The resulting code blocks can either be directly executed or passed to a *relinking* procedure.

Relinking In this step the previously generated code blocks are linked, if possible. To achieve better performance, the *relinking* [5] can make use of previously compiled blocks and produce in the best case a long phase of uninterrupted program execution.

Execution The execution of the recompiled binary code is straight forward. All critical operations have been replaced and only conditional jumps, exceptions such as page faults and illegal operations, or interrupts might alter the instruction flow. In such conditions, the code block is typically discarded and recreated, starting from the faulting instruction and extended with exception or interrupt handling routines. A different approach is to directly jump to a suitable handling function and continue afterwards. Once the available blocks have finished execution, the translator typically adds them to a translation cache and determines the new *instruction pointer* and starts decoding from this point.

2.3.2 Full-System Simulators

A variety of full-system simulators exist. They are designed to simulate different aspects of a system in more or less detail. Simulators vary from emulation – *functional simulation* – at the instruction level to provide *functional correctness* to timing accurate *micro-architectural simulation*. The following list of simulators is not complete, since various variations exist and many different research

projects [37, 39] have extended or adapted them to their needs. We therefore give an overview of the most commonly used simulators.

Bochs Bochs [28] was one of the first freely available system simulators, providing the facilities to run unmodified x86-compatible operating systems on different CPU architectures. To allow these OSes to run, Bochs simulates PC-compatible hardware. However, due to its interpretive mode, each instruction's equivalent is directly executed after disassembly. Bochs thus remains a factor of 30x slower than other simulators using DBT [5]. In general, it is a slow virtual machine, providing functional correctness, but no timing accuracy or micro-architectural simulation.

Gem5 Gem5 is a fusion of the M5 and the GEMS simulator and combines M5's CPU and interconnection model with GEMS's memory and cache coherent protocol simulation [6]. It forms a highly flexible simulator with a fine to coarse grained simulation. Gem5 is capable of emulating different ISAs such as ARM or x86, on different host architectures. It offers two operation modes: A *system-call emulation* only suitable for user land programs and a *full-system simulation*. The system-call emulation is very high level and does only intercept system service requests, similar to a *sandbox* and is therefore limited. The full-system mode presents a bare machine with a wide range of emulated devices necessary to run commodity operating systems, such as Linux. This mode of Gem5 is slower compared to other simulators, since it does not use a sophisticated binary translator [6]. Instead, it creates a list of virtual calls to emulate functionality of each disassembled instruction. Although, this approach is faster than a pure interpretation due to saving repeated disassembling, it cannot compete with full DBT; especially if a functional simulation is sufficient for a particular analysis. However, an advantage lies in its flexibility in terms of simulation detail and extensibility.

MARSSx86 In contrast to the previously described simulators, MARSSx86 [39] utilizes a dynamic binary translator and augments its code with cache, pipeline, in- and out-of-order CPU simulation functionality. It provides support of single- as well as multi-core configurations. It simulates complete cache hierarchies down to the cache synchronization protocol level, similar to the micro-architectural simulation mode of Gem5, but combined with a faster instruction simulation. Although it is based on the open-source simulator QEMU, it is far more flexible and allows switching from a fast result driven simulation to a more complex micro architectural accurate mode. However, it is only available for x86.

WindRiver Simics WindRiver Simics [31] is a commercial simulator, used in various scenarios, from debugging faulty programs, to malware analysis, and hardware simulation. It allows deterministic, cycle accurate simulations and supports various, flexible hardware configurations. Simics supports binary translation for various instruction sets, generic parametrized cache simulations, and runtime inspection of CPU and memory. The simulation guarantees functional correctness with optional accurate timing. A pipeline simulation, as in MARSSx86, is omitted for performance reasons, CPU models comprising a pipeline simulation can however conceptionally be added. The performance of an un-introspected function level simulation varies from 2.1 MIPS to 5.7 MIPS on an ancient Pentium III processor [31]. On recent hardware, intercepting every memory access leads to a simulation speed of about 3 MIPS [42].

2.3.3 Quick Emulator (QEMU)

In contrast to the previous described simulators, *Quick Emulator* (QEMU) [5] is widely used for virtualization in Linux environments. As hardware-virtualization became available on a wide range of processor architectures, the DBT is only used to simulate a different architecture, e.g., ARM on x86. QEMU is often used in conjunction with the *Kernel Virtual Machine* (KVM), but just provides emulated I/O devices. The instruction stream is directly executed on the processor and only privileged operations fallback into QEMU for handling. QEMU still comprises a

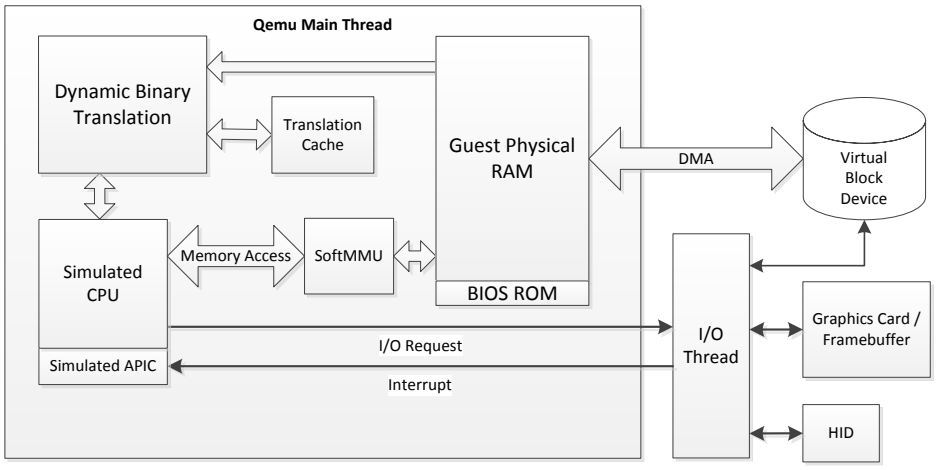

Figure 2.5: The organization of QEMU. It consists of two threads, one for CPU simulation and one for device emulation. The interaction is similar to a real system, but interrupt delivery can be postponed.

dynamic binary translation (*Tiny Code Generator* (TCG)) and can be extended to provide similar simulation features as present in Gem5 or Simics [37, 39].

The simulation mode of QEMU uses DBT with an *intermediate representation* (IR), to translate the guest ISA to the host ISA. The IR enables QEMU to support a sub-set, common denominator of all ISAs. The used intermediate representation consists of *micro operations*, a subset of all possible instructions of different architectures.

Figure 2.5 shows the general organization and operation of QEMU. QEMU provides two threads of execution. The main thread simulates the CPU state and executes the resulting code from the DBT. If QEMU simulates multi-core CPUs it schedules them to this thread in a round robin fashion, avoiding complex synchronization and locking mechanisms. A second thread handles DMA and I/O requests and triggers interrupts on completion, which will be processed after every successfully executed code block. QEMU emulates the behavior of real hardware, with a DMA controller and asynchronous operations, but with a slightly different timing, since interrupts can be postponed for longer periods, as they would do in a real system.

Chapter 3

Analysis

Previous research has been concentrated on pages related to the Linux page cache and I/O operations, since these pages are a primary source for duplicate page content in a virtualized environment [9, 35, 36]. The *page cache* buffers content read from or stored to block devices. Thus, in a virtualized environment identical page content resides in guest and host page caches. Existing page deduplication approaches either prevent duplication proactively, e.g., *Satori* [36], or merge the replicas instantly, e.g., *XLH* [35]. The hypervisor can easily collect page frames associated with the guest page cache by simply monitoring I/O operations – i.e., data transfers from and to virtual block devices – even without paravirtualization. That is because every I/O operation must be handled in the hypervisor or the host OS. It is an example of how information about pages can help to focus on promising memory regions and how these information can be collected.

However, there might be even more sources, which can easily be harvested to increase the efficiency and effectiveness of memory scanners. Although *XLH* and *Singleton* exploit nearly all sharing opportunities stemming from the guest page caches up to five times faster than regular memory scanners [35], they still cannot merge other sharing opportunities with similar speed. Their memory scanners still consume a large amount of time and computational power trying to identify and merge identical pages originating from *anonymous guest memory*. As suggested [41], some pages (e.g., about 54% of pages in desktop workloads) even change too fast to be harvested through periodical scans at all, while at the same time they are stable long enough to justify merging. Scanning with higher scan rates and more computational power might already solve some of these problems, but as memory can always be used for better purposes, the same is true for CPU resources; including the energy they consume when active. A direct hint to fetch moderate changing pages as quickly as possible or to exclude unstable pages could concentrate memory scanners on more promising regions and increase their effectiveness beyond what solely I/O focused scanners achieve today. To avoid ex-

pensive and futile scans, a restricted scan domain, with a high sharing probability is advisable. Otherwise, all pages of a VM have to be considered during scanning, even if a region contains stacks or virtual device memory and the probability of a *stable* memory page is low. Non-paravirtualized VMs lack semantic information and a memory scanner can therefore not exclude memory regions based on their internal allocation.

There are different ways to bridge this semantic gap. Firstly, a paravirtualized OS can be used to report allocation changes and other relevant system events. Secondly, techniques for *VM migration* and *memory reclamation* can collect similar information without modifying the guest OS. One way is to analyze the VM guest memory, searching for memory management structures and parsing the contained information [12]. Another way, is taken by XLH [35], as it monitors I/O operations to and from virtual block devices inside a hypervisor to identify and prioritize a VM's page cache pages. Another source of information are page tables, e.g., dirty bits might indicate changing page contents. Additionally, a statistical analysis might bridge the semantic gap as well and provide (workload specific) heuristics, minimizing the scan regions or prioritizing regions with high sharing probabilities.

As previous work showed, if a scanner focuses on pages with high sharing potential [35, 44], it achieves better and faster memory footprint reduction. However, there are still pages, which are shareable, but cannot be harvested due to slow scanning. We aim at discovering correlations between *duplicate memory pages* and statistical information to either focus on stable regions or to exclude unsuitable pages, *permanently* or *temporarily* from the scanning process and thus increase the effectiveness of memory scanners.

3.1 Analyzing Sharing Opportunities

We focus our research on sharing opportunities not yet harvested by a hinting mechanism, and thus, if merged at all, need to be fetched by a periodical memory scan. A requirement for merging sharing opportunities is *stability*. Otherwise, the likeliness of breaking a merged page is very high. Less is known about other properties of these remaining sharing opportunities.

Sharing Opportunities As we want to focus a memory scanner to unused sharing potentials, at first we have to detect all *sharing opportunities* in examined workloads. Without this quantification of the overall sharing potential an analysis of their properties is impossible. Sharing analyses, comparing every page's content with every other page, can be done in two ways. They can consider only

a single system and detect content duplicates within its own domain – an *intra-domain sharing*. Another way is to perform an *inter-domain sharing analysis*. It considers not only a single system, but also many systems, e.g., all VMs on a host, and detects sharing opportunities in all systems, which drastically increases the available sharing potential [41].

Semantic Information As shown in previous research [35, 36, 41], *semantic information* can help to provide better hints for a memory scanner. If a memory deduplication system is aware of a page frame's allocation, it can easily decide whether it should consider it during a scan, or leave it unexamined. For example, a scanner would be able to exclude stacks and virtual device memory permanently from its scans, as these regions have shown no sharing potential in the past [41].

Memory Access Frequency A frequently written page frame is an unsuitable candidate for memory sharing. Even if a sharing partner exists, it is most likely that the content will change in the near future and an already established sharing would break. Thus, a write access frequency for every page frame might help to determine pages, which should be skipped (at least for one scan pass), as long as they are changing fast. Low frequencies indicate probably more stable page frames.

Memory Access Patterns Extending the idea of memory access frequencies, it is possible that a distinct *memory access pattern* directly indicates the usability of a page frame for sharing. Memory access patterns use the temporal or spatial distribution of memory write accesses to provide a more complex identification scheme. It could be possible that different write access patterns directly lead to *stable* and shareable pages, whereas other patterns indicate the contrary. It is unclear, if more information about the history of each page helps to identify stable regions, or if access patterns show no statistically significant correlation to page (in-)stability. If such correlations exist and a scanner discovers such patterns, it might focus its scan operations on these pages, or skip them.

Semantics and Stability The conjunction of *semantic information* and *stability* of a page frame should improve scanning even further. Thus, as already shown, page cache pages remain stable for long periods of time [36, 41], the same might be true for, e.g., heap pages with a specific write frequency, access pattern, or write history. We have to examine, if and to what extend such correlations exist and if they can be used to improve memory scanning.

Write Working Set A working set describes the amount of accessed memory pages in a time interval [14]. It indicates the amount of memory and the pages used within the last observed period. As working sets are often estimated to determine, which pages can be evicted from physical memory and moved to swap space, it might also be useful for a memory scanner. We slightly modify this idea and consider only modified pages, which form a *write working set*. We exclude read pages, since reading a page does not change its stability, nor its sharability. Only write operations change content, and therefore break shared pages. If a page leaves a write working set, it could be considered for a memory scan, as we assume it to remain stable, i.e., unmodified. Afterwards for a sufficiently long time period to justify examination by a memory scanner.

3.2 Required Analysis Data

To verify if memory access patterns and stability criteria for duplicate memory pages exist, we have to collect different data for analysis. First of all, we have to ensure, we can identify *sharing opportunities*. Therefore, we have to collect memory content of the analyzed system. Information on memory accesses build the foundation for analyses of memory access frequencies and patterns as well as page stability. In contrast to reads, write operations may invalidate the information a scanner has about pages or even leads to breaking already shared pages. In consequence, at least every memory modification needs to be tracked.

Memory Content As program execution progresses memory content is modified. To be able to analyze sharing opportunities, a consistent memory image for every point in time must be available. Otherwise, shareable page content might be missed. *Memory content* is modified in two ways. Firstly, a program running on a CPU issues store operations, which change the content of main memory. Secondly, devices utilize *direct memory access* (DMA) to move content to and from main memory. To maintain a consistent memory view, these two operations must be traced. Monitoring these operations provides us also with the information necessary to analyze the memory access frequency, write access patterns and the access history of each page frame.

Memory Semantics To allow a more thorough and detailed analysis, we require additional *semantic information* associated with each page frame. Every page and its corresponding page frame have semantics associated with them. Most prominently, each page is allocated for a specific purpose, for instance to serve as a page for *named* or *anonymous* memory. The *allocation* of a page is important, but it can be further enriched with *mapping information* and *page table flags*. Only

this combination allows us to tell stacks and heaps apart, since both are allocated as anonymous memory. To further understand *memory access patterns* it might be helpful to distinguish the source of a memory operation, i.e., if it has been issued by the kernel or by a user-land program.

System State To trace store operations back to the issuing address space and in conjunction with scheduling information even back to the issuing thread, it is necessary to consider the current state of the examined system's OS. We, therefore, consider all *address space events*, regardless whether an address space is created, modified, or destroyed. The same is true for *task creation and destruction*. Furthermore, we want to track *kernel events* to be able to correlate different store operations to the dedicated kernel sub-system.

To improve memory scanning, we must identify new hinting sources or patterns, which allow to reduce the scanning overhead, by including or excluding pages. We came to terms that at least the information in Table 3.1, must be considered for our analyses.

3.3 Data Acquisition

To analyze sharing opportunities, page stability, and memory access patterns, all data for these analyses should be captured on a *non-interfering* level to avoid distortions of the examined workloads. As our analyses require different data sources, they should be collected in a single pass. Otherwise, correlation of semantic data, with memory accesses becomes difficult or might be impossible.

To measure the sharing potential some approaches regularly dump the memory of (hardware accelerated) VMs [35, 36], but typically that changes guest timing, affects the CPU caches, and scheduling policies. In the worst case it changes the complete behavior of a workload and cannot produce reliable data for analyses. Furthermore, memory dumps require large amounts of disk space and suffer from low temporal resolution. In the case of *Satori*, their snapshot resolution was only 30 seconds [36]. In contrast to *Satori*, we require every memory modification to calculate *memory access frequencies* in a high temporal resolution, not only long lasting sharing opportunities. To acquire memory accesses different methods come to mind.

One way to collect memory accesses is to map pages inaccessible in every referenced page table and modify the page-fault handler to trace read or write accesses. Then, a faulting instruction must be restarted in a single-step execution mode, with a temporally accessible page to perform the actual operation.

Purpose	Data to collect
Memory Content	• Memory Writes • DMA Writes
Memory Semantics	• Memory Accessor (Kernel or User space) • Guest Memory Allocations • Mapping Information • Page Flags
System State	• Task Creation and Destruction • Scheduling Information • Address Space Creation, Modification, Destruction • Kernel Events

Table 3.1: The three major data categories for a thorough, semantically enriched, sharing opportunity and pattern analysis.

Immediately after this operation the debug-exception handler can reset the page table bits to mark it inaccessible. These modifications are not only very complex and come with a high overhead, they also require a disassembler to decode the faulting instruction and to determine the performed operation. Another way is to utilize performance counters, which issue an interrupt every time a memory access occurs which is processed by a handler routine. This routine must analyze the operation, but might not be able to determine the actual written data, since it happens after a memory access. These two methods are not only expensive due to overhead, they typically miss memory accesses, e.g., from DMA capable devices. However, a VM might be able to ensure that no accesses are missed, but that implies a radical change of the hosting OS and the hypervisor. Furthermore, it still disturbs scheduling, guest timing, and CPU caches. A better and more reliable way to retrieve memory content without distortions, is a *simulation*.

Previous research clearly showed the capabilities of *full-system simulators* and their advantages over other data acquisition methods (for a detailed analysis of different techniques see [41]). In contrast to a hardware-accelerated VM, tracing memory accesses is simple in a full-system simulator and does not affect the timing, execution order, or scheduling within the guest system, at all. We therefore decided to use a *full-system* simulation.

A great drawback of this approach is the slow execution of simulated systems. They tend to be slower by magnitudes than real systems [42]. As they are already slow – even without complex introspection – a data analysis during simulation becomes even more expensive and reduces flexibility as online analyses require a rerun of a simulation for every considered question. Therefore, and to allow a simple correlation of different analyses, it is best to only collect data during simulation, save it and process it offline.
A full-system simulation can simulate various aspects of a system and recording all information is unfeasible. As we focus on memory access patterns and semantic information, we only collect all necessary information. The required data fits into three groups. Firstly, *direct recordable data* produced by a simulation, e.g., memory accesses. Secondly, *semantic information* from within a guest OS, such as memory allocations. The third group consists of data which can be inferred from *memory content* or *semantic information* by an analysis, e.g., *sharing opportunities* and must therefore not be recorded explicitly.

3.3.1 Memory Content

The first category of data is sufficient to analyze *memory access frequencies* and *memory patterns*, or to infer *sharing opportunities*. In a full-system simulator, memory accesses are traceable without any change of operating system code. This important data can therefore be collected independent of guest OSes, workloads, and simulated ISAs.

In contrast to many previous works [4, 35, 36], we cannot rely on periodical memory dumps. Although dumps are sufficient to provide sampled statistics about the sharing potential in a system, they are not a basis we can draw conclusions about the memory access frequencies or patterns on. That is because dumps only capture memory content, but not modifications, and write operations must not lead to changed content[1]. We therefore need to instrument the simulation to provide information on every write operation to be able to analyze access frequencies and access patterns afterwards. At the same time, we still need to be able to find sharing opportunities based on these information.

[1] Previous work reported that up to 34% of write accesses do not change memory [41].

There are two ways to store memory modifications: Firstly, like in previous work [41], after each memory modification event, a hash can be calculated for the target page frame and stored for later retrieval. In that case, identifying sharing potential is based on page hash comparisons. Secondly, memory operations can be traced directly with destination address and data. That allows to replay every memory modification and recreate a complete memory image. Both methods are in general suitable to base our measurements on.

The first method is inspired by previous studies on memory duplication, which are mostly done with hashes or checksums [1, 36, 41]. That suggests to save page content hashes and avoid a memory content rebuild during analyses. Although it seems to speed-up sharing opportunity analysis, this method has two major drawbacks. Firstly, a suitable hash, with nearly no hash collisions, consumes at least the same amount of memory as the data actually written by a memory operation. Secondly, a hash calculation for every memory modification slows down simulations by a magnitude. A fast hash function such as CRC32 on the other hand, is not sufficient to avoid *false positives* – i.e., wrongly identified duplicate page content – during recording. Although a *cryptographic hash function* or larger hash value might minimize this probability, it is harder to compress, due to its higher entropy and thus less suited to be recorded.

The second method stores every write with its destination address and the actual written data. Thus, the complete memory content can be restored and different questions can be answered with the same trace data. As stored write operations can rebuild the complete memory content on different temporal and spatial resolutions, they allow flexible inter-domain sharing analysis, and furthermore, an extraction of instructions and loaded file content if needed. Identifying sharing potential from a reconstructed memory image is trivial and precise. A drawback of a memory rebuild is however, that the analysis requires at least the same amount of memory as the examined simulation did. We conclude that storing memory writes directly, will provide most flexible trace data for an offline analysis.

A subsumed comparison of both methods can be found in Table 3.2.

3.3.2 Memory Semantics and System State

As mentioned earlier, important semantic information, e.g., memory allocations, must be recorded. It is not directly included in a memory image and complicated to extract with tools, such as *crash* [2] or *Volatilitux* [17]. Such an extraction from a reconstructed memory image, can be erroneous due to missing symbols or partially written data structures. Furthermore, it is slow. Directly recorded *operating system events* allow to reconstruct the system state faster. Therefore, we explicitly record *memory allocations* and *system events* changing the current *system state*

Stored Data	Pro	Contra
Hash	Fast Sharing Analysis	False Positives High Entropy Low Flexibility
Actual Data	**Fast Simulation** **Good Compression** **High Flexibility** **Memory Content**	**High Memory Utilization**

Table 3.2: A comparison of how to store memory modifications. Storing actual written data is fast and allows most flexible analyses. Our chosen method is in bold.

from within the analyzed OS. That does not only reduce the reconstruction overhead, it also allows flexible OS introspection.

3.4 Simulation

Many different simulators are available with a coarse- to fine-grained simulation detail, providing *function level* to *micro-architectural* simulation modes. As a *micro-architectural* simulation must simulate, for instance, a CPU pipeline, the memory bus, and cache coherency protocols, it has to execute more code per simulated instruction, whereas a *functional* simulation can just simulate the result of an operation. Thus a functional simulation is faster. We focus on execution speed, since previous work [25, 41] clearly showed simulation speed as a bottleneck in the analyses process. If simulations take up to weeks, and making only little progress, simulating complex and long-running workloads becomes unfeasible. Rittinghaus et al. compared different simulators and measured their execution speed, which ranges from 3 MIPS up to 93 MIPS for memory inspection [42].

There are three main goals, which a simulator should provide. Firstly, it should be able to allow memory inspection and pass information from within the guest OS to the tracing infrastructure. Secondly, it should be able to provide an accurate timing source to correlate system events with memory accesses. Thirdly, the simulation should run as fast as possible.

3.4.1 Full-System Simulator Comparison

All described simulators in Chapter 2 are, in general, suitable for our research, but they greatly differ in *adaptability* and *extensibility*, *simulation detail*, and *execution speed*. A suitable simulator should provide necessary interfaces to intercept memory accesses and record system events. For this purpose a functional correct simulation should be sufficient. All considered simulators already provide these interfaces or can be extended to do so. Thus, the main goal is to find a fast simulator to allow long-running and complex workloads, e.g., SPEC benchmarks, to complete within reasonable time. With a far too complex simulation – simulating processor pipelines or cache hierarchies – such workloads run up to weeks [25]. As the implemented functions directly affect the maximal achievable simulation speed, a simulator has either to provide a flexible configuration interface or source code; thus, allowing to modify the simulation, the simulation level, and its details. If that is not the case, the simulator is most likely unsuitable for our purposes.

Most commercial simulators, e.g., Simics, provide a rich introspection interface with a wide-range of simulation features. As Simics is used for different kinds of simulations, for debugging, for development of new hardware, and for malware analysis, it provides many features, which are gratuitous for our analyses. For instance, Simics provides only a single hook for memory operations. Although, we are only interested in write accesses, the hook is called for every load operation and thus unnecessarily reduces simulation speed. On each hook invocation Simics collects a variety of information, e.g., flags from the page tables and page attribute tables, to ease access to these values. That, however, introduces an extra overhead, whereas we only need such attributes, if they actually change and not for every write operation. Furthermore, each memory access is modeled as an object for a timing accurate cache simulation, which decreases simulation speed even further. Simics does not provide interfaces to change this behavior and as source-code is not available, we cannot adapt it and increase its execution speed. A simulation with active memory inspection progresses with 3 MIPS [42] and is far too slow, although it utilizes *dynamic binary translation* (DBT).

Open-source simulators, such as MARSSx86 and Gem5 allow to simulate complex systems and support inspection. These primarily *micro-architectural* simulators with multi-core, pipeline, and cache coherency protocol support, provide a far too complex and slow simulation. The details are superficial for our memory analyses. Both MARSSx86 and Gem5 allow to switch between *micro-architectural* and a fast-forward *functional* simulation mode during runtime. In addition, Gem5 provides an extensible interfaces and can be configured without complex source-code changes. The *functional* simulation in addition with a flexible configuration interface would make Gem5 a suitable candidate. However, its primary weakness, is the lack of a full dynamic binary translation [6]. Instead, it

uses a list of virtual function calls, which decreases the maximal achievable speed by a magnitude. That leaves only two remaining simulators for our data recording.

Even without a configuration interface open-source simulators have one advantage over commercial simulators; gratuitous features can be deactivated, simplified, or adapted to meet specific needs. Although, available source code does not ensure an easy extension [20], it is possible. The major problem of many open-source *functional simulators* is their lack of inspection and introspection hooks. As Bochs and QEMU were originally designed to emulate hardware and provide an early implementation of virtual machines for x86 architecture, they completely miss memory inspection and tracing capabilities.

Bochs cannot be used, since it interprets every instruction and is therefore very slow [5] – 30x slower than QEMU. Thus, the only remaining candidate for data acquisition is QEMU. Although, it does not provide the necessary interfaces, nor the inspection functionality on *instruction level* – required for memory hooks and OS introspection, it is fast and has been extended in various research projects [37, 39] in the past. Rittinghaus et al. claim that the simulation speed of QEMU is about 93 MIPS [42] even with implemented memory inspection.

Considering the different aspects of various simulators and the requirements for our data acquisition, QEMU appears to be well suited. It is *open-source* and extensible, in contrast to other simulators, it is *fast*, and data acquisition should not take weeks to complete. And as *functional correctness* suffices for memory inspection and OS introspection, we chose QEMU as basis for further modifications. Table 3.3 summarizes the pros and cons of the considered simulators.

Simulator	Extensibility	Simulation Detail	Speed
QEMU	**Source**	**Functional**	**DBT - 93 MIPS**
Gem5	Source	Both Modes	Interpretive Translation
Simics	Commercial	Functional	DBT - 3 MIPS
Bochs	Source	Functional	Interpretive Mode
MARSSx86	Source	Both Modes	DBT

Table 3.3: Considered simulators sorted by their suitability for our data acquisition. The most suitable (QEMU) is mentioned first.

3.5 Conclusion

Different memory deduplication systems use memory scanning as the discovery mechanism for *sharing opportunities*. As they are unable to identify shareable content fast, some improvements using I/O hints have been achieved [35], but

they still cannot harvest all theoretically detected potential. We want to improve memory scanning even further, by either focusing on stable pages or excluding unstable pages temporarily or permanently.

To analyze properties of this remaining sharing potential, we want to quantify it and find correlations between stability of page frames with memory access frequencies and memory access patterns. To do that, it is necessary to trace all modifications of main memory. We, therefore, record all write accesses to each memory address and utilize a reconstruction of the full main memory image and its temporal progression to do the sharing and access pattern analyses. Additionally, we collect *semantic information* from within the guest OS. That allows a more thorough analysis of sharing opportunities and their properties, as we can link stability with internal allocations. To successfully correlate memory semantics with each page frame's content, we have to collect all data in *single pass*.

The main problem of tracing memory modification is the potential distortion of the examined workloads by the employed monitoring mechanism. To avoid distortions, we use a full-system simulation. The simulator only needs to provide interfaces to collect the analysis data, e.g., write operations, a timing source for correlation and simulation time, and functional correctness – a complex *micro-architectural* simulation is unnecessary. After data collection, *offline analyses* allow to answer questions about the correlation of sharing potential and page access characteristics.

Previous research showed that simulation speed can be a bottleneck, and prevents long-running workloads to be recorded. We favor a simulator with high simulation speed and low simulation overhead. Therefore, we have selected QEMU—a fast and extensible simulator. It is the basis for our data acquisition, although it lacks the required interfaces, it can, as previous work shows [37, 39], be extended and still retain a higher simulation pace than other simulators.

Chapter 4

Design

In this chapter, we discuss the design considerations. We review previous trace data acquisition, analyze its limitations and suggest improvements. A previous research project [41], which is strongly related to our research, is designed to analyze *sharing opportunities* in memory and CPU caches, and utilizes OS introspection to acquire semantic information. It is based on a two step model, (1) a full-system simulation with trace data recording and (2) a separate framework for offline data analysis. The data acquisition is based on Simics a full-system simulator. It records memory modifications on a hash basis and semantic information retrieved from a modified Linux kernel. This data is semantically grouped into sequential data *streams* and stored in a trace file. In a second step, the analysis framework retrieves the trace data and performs offline analyses on them.

As we want to examine sharing potential with semantic information and memory access patterns, we base our work on the proposed design and available analysis framework. Although many basic blocks of this design can be reused, e.g., the idea of stream oriented storage and the separation of simulation and analysis, many parts must be modified and improved to ensure that all analyses described in Chapter 3 can be performed.

A main problem of the previous design was the slow execution speed of the chosen simulator (Simics). It made simulation of long-running workloads infeasible. Therefore, the low simulation speed led to relatively short traces and resulted in scalability problems when a faster simulator is employed that produces longer traces. To overcome these limitations and use QEMU some adaptions are necessary. We also had to re-implement parts of the offline analysis framework to avoid cumbersome analyses for our use case.

In the remainder of this work, we will refer to the design in [41] through the terms 'original' or 'previous design'.

4.1 General Design

In contrast to the original design, our design follows a three step process, (1) *system simulation* and *trace data collection*, (2) independent *data storage*, followed by (3) *offline data analysis*. Our overall design is shown in Figure 4.1.

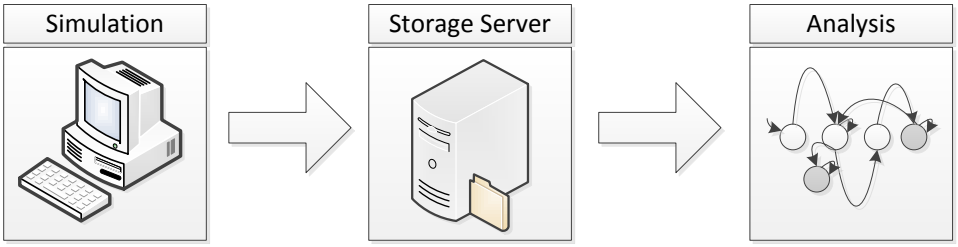

Figure 4.1: Our three step design. Data acquisition, storage and offline analysis.

We moved the saving and loading of trace data into a dedicated stage (i.e., component), because that made changing the storage format, which we needed to adapt, easier. It also contributed to a cleaner overall design. The component responsible for the second stage is the *storage server*. The server runs alongside the simulation, collects information about memory accesses and semantics and stores them on disk.

In the original design a minimal storage server had already been introduced to overcome address space limitations of a 32 bit simulator, and thus, allowing more buffer space for trace data and asynchronous compression. However, the original server only handles saving traces and does not offer any interfaces for reading previously recorded data. Instead, the original design integrates a separate loader into the analysis framework.

We stuck to the fundamental idea of a storage server, but extended it to a full *data store*, which provides a simple storage interface for producers *and* consumers. It hides trace buffer management and storage details such as compression, pre- and post-processing, and integrity checks form all clients—i.e., the simulator and analysis tools. This way, producers and consumers are able to store or load trace data without considering the actual background storage and its format. To upgrade the previous storage server design, we had to improve portability, compatibility, and scalability as well.

In the original design, the Windows version of Simics had been used to gather all data required for analysis. For that reason, the previous tracing and storage components had been tailored to run on Windows through the use of OS-specific APIs and mechanisms (e.g., I/O completion ports). As mentioned in Section 3.4.1, we chose to base our simulations on QEMU. Although QEMU can be built for Windows, the process is cumbersome and the OS support is still in an alpha stage. We therefore decided to base our work on the Linux version of QEMU, which is currently the best supported and stable one. Since we also wanted to maintain compatibility with Windows to be able to use the Windows-based analysis tools already present in the previous design, we decided to adapt the storage server to be OS independent. We achieved that by moving OS-specific components in the storage server into a respective abstraction layer. In its present form, the layer supports newer versions of Windows (XP+) as well as POSIX-compatible operating systems.

Besides changing the operating system, moving from Simics to QEMU as simulator also required changes in the interface exposed by the storage server. To ensure compatibility to other simulators, we decided to provide a generic interface to the storage server. It exposes buffer space via shared memory for trace data collection and RPC communication for interaction between client and server. These two mechanisms are encapsulated in a small library, allowing an easy integration into a client. The library and the storage server manages data storage and communication. The client just needs a minimal *integration layer* to write trace data into the supplied buffers.

Further changes were necessary to increase the scalability of the storage format, the details can be found in the next section.

4.2 Trace Organization and Storage

In the original design, Simics was used as a simulator, but due to its slow simulation pace only short workloads could be examined. The resulting trace files are comparatively small with 30 GiB (compressed). We replaced Simics with QEMU to focus on long-running workloads, therefore we expect more data and larger trace files. Initial experiments showed that the traces for our evaluation can easily grow beyond 500 GiB (compressed). However, the original file format does not offer adequate scalability and we had to change some crucial aspects.

An original trace file, as seen in Figure 4.2, consists of a linked list [41]. This linked list is conceptionally divided into *streams* with variable-sized trace-lists containing variable-sized and compressed *trace entries*. We stick to this concept

and continue to use *streams* and *segments* (previously called trace-lists). The main scalability problem originates from a missing indexing scheme and variable-sized trace entries. Thus, scanning a file is necessary to access contained trace data.

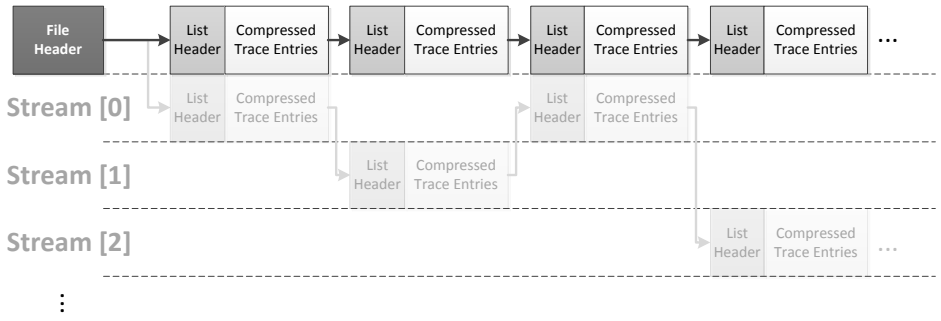

Figure 4.2: The previous trace file format stores a linked list of variable-sized trace-lists. Each list comprises a header and compressed trace entries. The lists are tagged to associate them with the corresponding streams [41].

Instead of variable-sized structures, we use fixed-sized trace entries to store the recorded data. Trace entries are grouped into fixed-sized *segments* to allow random access to trace data through segment-wise decompression.

In contrast to the previous design, mixing trace entry types is not allowed. That enables us to access an entry in $O(1)$ simply by its index. Each segment in turn is associated with a dedicated *stream*, which partitions data according to its semantic connection and type of information. We added an indexing scheme, which allows direct access to every segment in a data stream, without pre-scanning the complete trace file. Figure 4.3 shows the indexing scheme and the structural dependencies of trace entries, segments, and streams.

Segment The smallest allocation and management unit is a segment. It is managed by the storage server and presented to each client through a middle-ware API as a contiguous memory buffer. Segments store the actual data in the granularity of fixed-sized and probably compressed *trace entries*.

Stream Segments form linear data streams. Each stream is exposed to a simulation, as to an analysis client and helps to structure logically and semantically linked data. It further allows to associate meta-information with stored trace data. For example, every system event of a guest OS has its own dedicated data stream,

which allows more flexible access as in the original design. For an analysis all related streams can be opened without the need to parse unrelated data, which was originally necessary to de-multiplex the trace-lists.

The original file format lacked a fast indexing scheme, which forced the analysis to pre-scan the complete file to discover included streams, segments, and the quantity of trace entries. It took up to 2 minutes to scan a 30 GiB trace (approximately 300 GiB uncompressed data) . That initial discovery phase makes working with long-running traces cumbersome. Therefore, we augmented the storage file format, which is only visible to the storage server components and not exposed to producers or clients, with a fast lookup scheme, to access any stream inside, and every containing entry with less overhead.

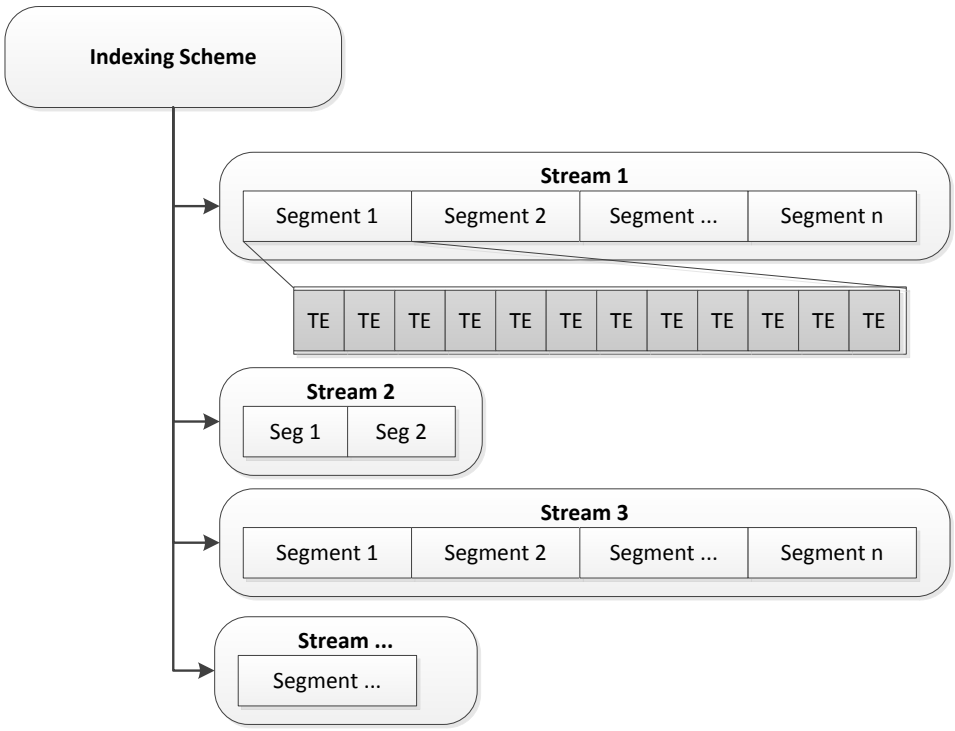

Figure 4.3: A trace store consists of streams. Each stream consists of segments, which contain fixed-size trace entries. A fast indexing scheme avoids cumbersome data discovery on trace loading.

4.2.1 Analysis

The original analysis framework is responsible for correlating the recorded sharing opportunities, instead of full memory writes, with a reconstructed system state of the simulated guest OS and contained semantic information. To achieve that, the framework takes the trace data from the OS introspection and rebuilds the currently running processes, their associated address spaces with their internal layout and allocations of each page frame. To determine the running process associated with each memory event it further replays scheduling. The reconstruction runs in parallel as depicted in Figure 4.4. For every point in time, such state can be reconstructed and used to enrich memory events and to provide detailed usage information for each page frame. We stuck to this fundamental principle, but allow to reconstruct only parts of the system state, e.g., only memory allocations. That is necessary to increases scalability and flexibility. Further extensions were necessary to deal with 64 bit OS environments.

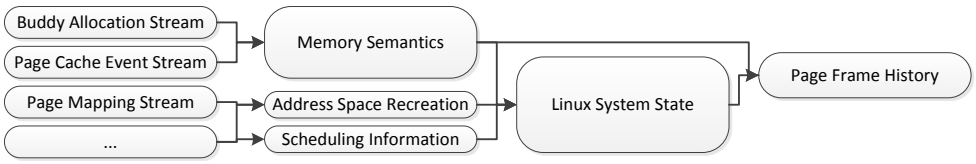

Figure 4.4: The recovery of a OS system state, based on various system event streams.

Memory Reconstruction In the original design, all sharing opportunities were detected during simulation and a corresponding hash value was recorded. We came to terms that recording write operations directly with their associated data makes sharing analyses more flexible. Firstly, it allows to infer sharing opportunities in different temporal resolutions. For example, only after 100 ms, as shorter living sharings can hardly be exploited. Secondly, to analyze inter-domain sharing potentials. If sharing opportunities are detected during simulation, inter-domain shareable content cannot be identified correctly. Thus, a full memory image for every point in time is necessary. The reconstructed memory content of different recorded traces allows flexible and false positive free inter-domain sharing analysis. Thirdly, as we want to analyze memory access frequencies and access patterns, they can be easily tracked during the reconstruction process.

Thus, our revised design moves everything except the actual recording from the simulation to the analysis framework. This strict separation is necessary to increase simulation speed and gain the described flexibility. Our framework exten-

sion is capable of replaying memory modifications and rebuild the corresponding memory image. The replayed data consists of a sequence of write operations to the *guest physical memory*, either performed by *direct memory accesses* (DMA) from external devices or by accesses within the current control flow. Alongside the reconstruction *sharing opportunities* for a predefined time interval, are analyzed and statistics for memory access frequency and pattern analysis are gathered.

State Compactification The memory reconstruction already provides necessary information, like *sharing opportunities* and a *memory access frequency*, but to analyze *memory access patterns* and *stability* a single memory image is not sufficient, instead a complete history for each page frame must be retrieved. Additionally, information from the reconstructed system state is missing, to correlate memory access patterns with memory allocations. Since storing the system state for every write operation is expensive, a compact representation for access histories is necessary. We propose a list of bit fields, which comprises write frequency, mapping information, memory allocations (e.g., to the page cache), and the results of the sharing opportunity analysis. Although the memory content is stripped, this representation, still consumes large amounts of memory. To further reduce memory consumption an optional sampling step can scale the state history down.

Figure 4.5 shows the fusion of sharing analysis results with the information of the system state recovery – most important the page frame allocations – to a compact memory system state.

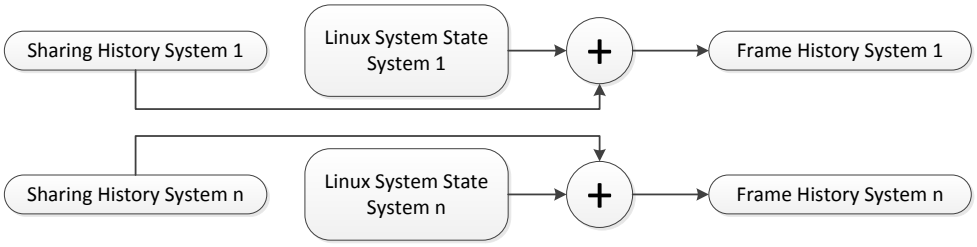

Figure 4.5: The recovered OS system state is fused with the information from the memory reconstruction, to form a compact page frame history.

Simulation Adaptation As we concluded in Chapter 3, we utilize QEMU as full-system simulator for data acquisition. However, QEMU is missing essential features to intercept memory accesses, introspect guest OSes, or to provide reliable simulation time. To support our trace data requirements, we must add three additional interfaces to QEMU.

Firstly, QEMU does not allow to monitor memory modifications, but as we want to analyze sharing opportunities, memory access frequencies and patterns, it must be extended to intercept every write operation and DMA to main memory.

Secondly, QEMU does not provide an interface to transport data from specially modified guest OSes to the tracing infrastructure. As our analyses require system events to enrich page frames with allocation information, an additional handling mechanism is necessary.

Thirdly, the timing facilities provided by QEMU are limit and do not allow to get a notion of simulation time, but to analyze page frame stability and access frequencies such time source is indispensable.

4.3 Conclusion

For data acquisition and analysis we utilize the framework created by Rittinghaus. It is suitable for our purpose and provides valuable features, e.g., to reconstruct the recorded OS system state. However, some modifications and extensions are necessary to allow fast and scalable data acquisition and processing. As the whole framework was originally designed to trace data with Simics on Windows, we had to ensure portability of the storage component to do data acquisition with QEMU on Linux and still use the analysis framework on Windows.

Additionally, changes in the interfaces exposed by the extended storage server are necessary to ensure that it operates with different simulators, for this purpose a middle-ware API encapsulates communication and buffer management and exposes only a compact interface to the integration layer of QEMU.

The scalability of the original design was limited, as the file format can store, but not open the contained data within reasonable time. That would have made analysis cumbersome. We further expect even more trace data, as we examine long-running workloads. Hence, we redesigned the trace file format to allow fast indexing and direct access to all data segments during analyses.

In contrast to the previous design, we moved sharing opportunity analyses out of the simulation into the analysis framework and store actual write operations. Therefore, we have to implement a memory reconstruction that allows to rebuild memory content, track memory access frequencies, and detect sharing opportunities during offline analyses. To analyze temporal access patterns we further need the history of every page frame correlated with results of the OS state reconstruction.

As we moved our data acquisition from Simics to QEMU, additional inspection interfaces and a simulation time source must be added to provide the required data for memory content, frequency, and pattern analyses.

Chapter 5

Implementation

In the following sections we will describe the implementation details of the modifications made to the simulator and the original tool chain of Rittinghaus [41], which we use as basis for our work. We start by presenting detailed information on the extensions added to QEMU as source of trace data. We continue by explaining how we improved the saving and loading of traces and finish with an overview of the internals of trace data processing.

For the remainder of this work, we refer to the simulated system with the term 'guest'. We call the system executing the simulator 'host'.

5.1 QEMU Modifications

As our simulation front-end consists of QEMU a fast, open-source, full-system simulator with functional correctness, we had to augment it with the necessary trace hooks. QEMU's primary development goal was to provide simulated hardware to execute different OSes under various architectures, not to allow thorough inspection of memory accesses or operating system internals. It lacks monitoring features and an accurate simulation timing facility. Therefore, we have extended QEMU to provide monitoring features for memory store and load operations, accurate timing, and a hypercall interface to communicate semantic information from within the guest OS to our tracing infrastructure.

5.1.1 Memory Hooks

To gather information about memory accesses performed by the simulated system, an interception of memory operations is necessary. We found three different kinds of memory operations that are performed by QEMU. Firstly, the most frequent operations are load and store operations from within the CPU's instruction flow. The

second most frequent memory operations are *direct memory accesses* (DMA). The last group comprises memory operations performed by the emulation of complex CPU behavior such as page table modifications triggered by the MMU. We added interception points (hooks) for each group of memory operations, allowing us to trace their execution at runtime (see Figure 5.1). In the following we explain each hook in detail.

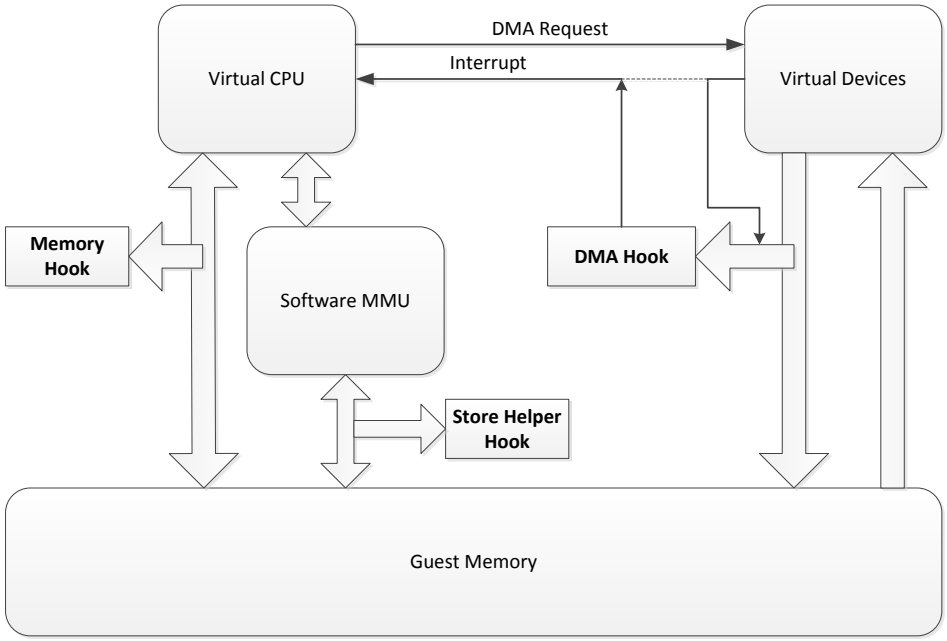

Figure 5.1: Overview over different memory hooks, as necessary to intercept main memory operations in QEMU. The memory hook is invoked most frequently.

Load / Store Hooks

The binary translation of QEMU emits load and store operations for every memory access of the virtual CPU. Our modification intercepts the code generation and appends a call to a hook procedure as shown in Figure 5.2.

QEMU's intermediate representation (IR[1]) facilitates this extension as each load and store operation maps to a respective single instruction in the IR. This

[1] See Section 2.3.3 for further details.

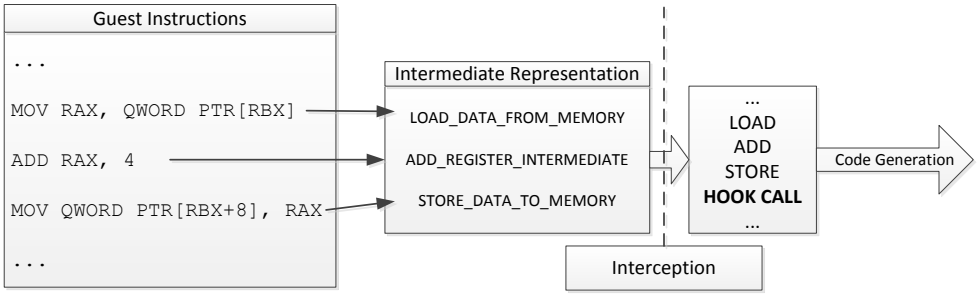

Figure 5.2: The binary translator disassembles guest instructions and maps them to a compact intermediate representation. Every intermediate load or store instruction is extended with a memory hook callback.

is true even for those memory operations implicitly performed by complex instructions such as a store on a stack invoked by a register push (e.g., `push eax`). The invocation of our hook is transparently added to QEMU's translation of IR instructions. In the hook we have access to the data size, address, and actually stored or retrieved data.

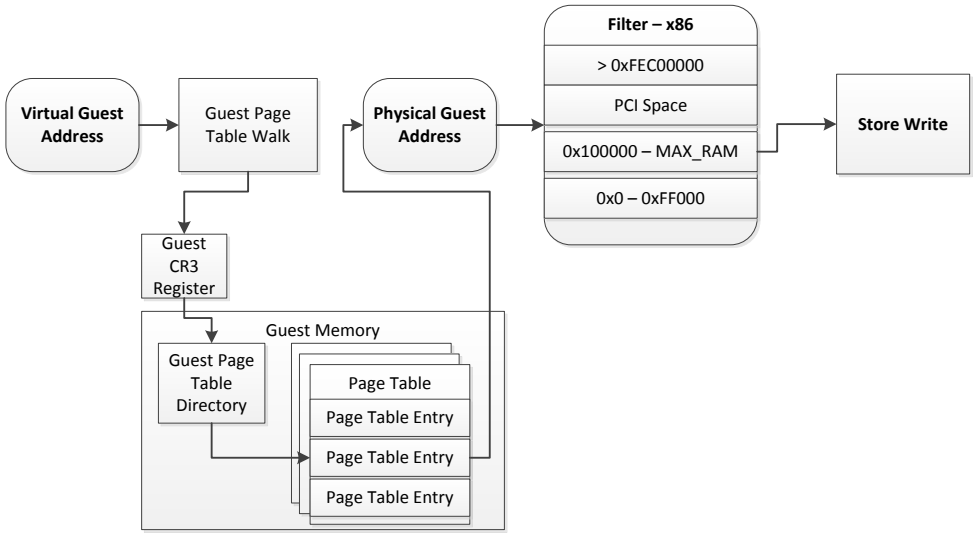

Figure 5.3: Memory operations use virtual guest addresses. For our analyses we translate these to their physical counterpart. Accesses on not interesting address ranges are ignored.

The address visible to the intermediate operation layer, and therefore visible to our memory hook, is a *guest virtual address*. As our analysis of sharing opportunities and the involved memory reconstruction work at the physical memory level, a translation to guest physical addresses is required. The translation itself is a regular walk of the currently active page table in guest memory. As we are not interested in memory accesses addressing hardware reserved ranges (e.g., PCI device memory), we exclude these accesses from the trace. Figure 5.3 depicts the address translation and filtering process.

DMA Hook

To reconstruct the memory content, an important data source is *Direct Memory Access* (DMA). More than two-thirds of all data stored in main memory are transferred via the DMA controller. Due to its asynchronous nature, DMA must be treated differently from operations in the regular control flow. Once the OS requires access to a specific device, it sets up a DMA transfer. QEMU intercepts this operation and locks the source and destination pages. Afterwards, it creates an I/O request packet, which is processed asynchronously in a different thread (see Section 2.3.3). On completion the thread raises a simulated interrupt. The

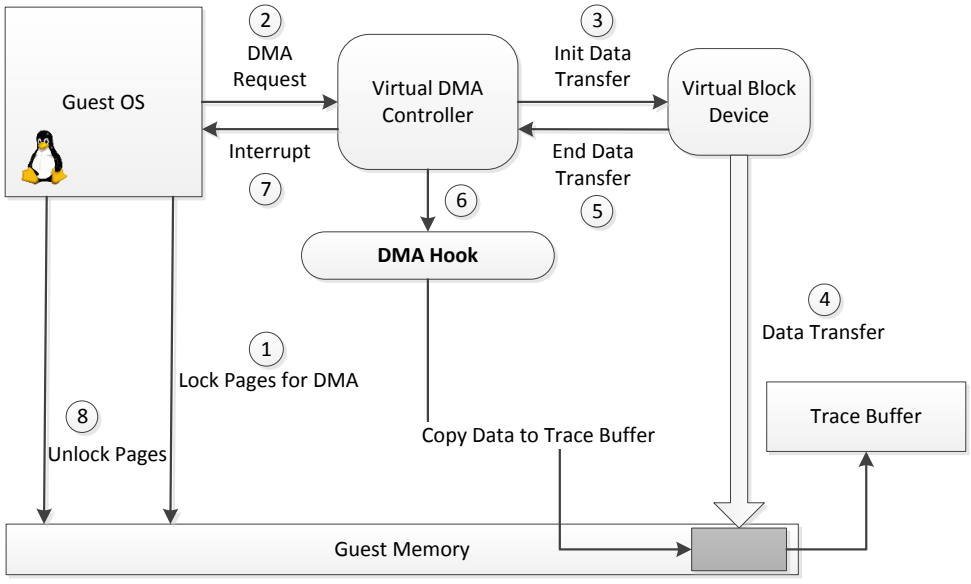

Figure 5.4: The typical DMA processing for storage devices in QEMU intercepted by our hook.

hook procedure is invoked right before the interrupt request is issued. In this stage the data has already been copied into guest memory, but the OS still does not consider the operation to be complete, and will therefore not modify the page frame. The hook copies the frame's content to a memory buffer supplied by the trace infrastructure; thus generating a corresponding entry in the trace that allows the DMA operation to be replayed in the memory reconstruction during offline analysis. A simplified DMA operation and the invocation of our hook is illustrated in Figure 5.4.

The timing of DMA operations in QEMU varies with the scheduling and the I/O processing in the host, and it might not be identical in every run. Thus, this part of QEMU's full-system simulation is indeterministic, in contrast to Simics. Every different run diverges as time progresses. We favor QEMU's behavior as the indeterministic timings allow more realistic workload execution.

Store Helper Hooks

Although the previous two hooks collect all memory accesses issued by a program or the guest OS, they are not sufficient to reconstruct the complete guest memory's content. After a reconstruction based on these two hooks, some page frames still have diverging content, if compared to a full memory dump. Therefore, we investigated all possible memory modifications issued by QEMU. This way, we found QEMU to use so-called *store helpers* to modify memory in the course of special system behavior—for instance, flip the dirty bit in page table entries from within the emulated MMU or push the CPU's execution context on exceptions. These helpers can hardly be found from pure source code studies, as they are created with complex pre-processor directives, a factor also discovered by other research projects [20]. We installed hooks in all relevant store helpers.

Store helpers are different from regular memory stores, in that they use host virtual addresses instead of guest virtual addresses. To be usable in the analysis, our hook makes a remapping to guest physical addresses. The store helpers are called alongside the regular instruction flow and do not increment the instruction counter, nor change the binary translation. From a guest's perspective these operations happen simultaneously to other instructions, as they most likely would do in a real system.

5.1.2 Instruction Counter

An important facility is a source for timestamps to correlate trace data within different data streams. However, even more important is a notion of time to evaluate *memory access frequencies* and page frame *stability*.

Timing Facility	Source	Accuracy
Wall clock time	Time-of-day clock of the host	Host time
Host cycle-counter	Real CPU Counter	Host Cycle Counter
ICounter	Executed Guest Instructions	Refreshed after I/O
Instruction Counter	**Executed Guest Instructions**	**Refreshed before read**

Table 5.1: Available time sources inside the simulation. Our implemented time source is in bold.

QEMU makes different timing facilities available for the guest OS and therefore for our hook procedures. A complete list can be found in Table 5.1.

Firstly, it passes the *wall clock* of the host as a time-of-day clock into the guest, which is unsuitable for guest accurate time measurement, even for an unmodified simulator. The time visible within the simulated system, will always reflect the scheduling of the host OS and show overhead caused by binary translation.

Secondly, the *host cycle-counter* can be used inside the guest as well; it is passed through as it is, so it does not hide the cycles consumed by instrumentation, binary translation, or even operations of different tasks in the host system. It might only be suitable as a source for random number generators.

Thirdly, QEMU has a simulated cycle-counter called *icounter*. It is a virtual instruction counter, counting only the guest instructions without considering the overhead caused by binary translation. That is a suitable source for guest accurate timing. However, for performance reasons, it is only updated during I/O operations and does not reflect the time in-between these periods. For our tracing, we require a counter, which operates on instruction granularity and is available and refreshed in every hook.

All in all, QEMU does not provide a compatible time source, which is suitable for trace data timestamps and correlated simulation time to real time. That made a new time source necessary, loosely based on the already available *icounter*. This new counter runs alongside the translated code and is updated before an invocation of a hook. To minimize the overhead, the code generator emits only a single instruction to refresh the counter. It allows accurate cycle counting, with only little runtime overhead. Every operation is counted once, and does not provide a micro-operation accuracy. This behavior is similar to the basic timing in Simics [31] and fully sufficient for our purposes.

5.1.3 Operating System Introspection

The memory hooks described in the last section enable us to track memory accesses, identify sharing opportunities after memory reconstruction, and infer sta-

tistical data about access frequencies and patterns. However, these results still miss semantic correlation, which requires information from the guest OS running within the simulation.

We modified the recent Linux kernel 3.9.2 to collect relevant semantic information, for allocations, page mappings, and scheduling events and report it to the tracer. A complete list of all recorded events can be found in Table 5.2. We extended the 64 bit (amd64) version of the kernel similar to previous work [41]. As the Linux kernel is regularly extended, internal data structures as well as interfaces change very often. We had to adapt the already available introspection from the original design to comply with new structures. For instance, the interface for page table manipulation was redesigned to ease porting Linux to new architectures.

To collect semantic information, the guest OS introspection requires an additional interface to communicate with the tracing infrastructure. QEMU does not provide any interfaces or a *magic instruction*, like for example Simics. A magic instruction has no function, except that it calls right into the simulator without altering the system state. With such an instruction a guest OS can pass information to the host. We refer to this simulator invocation as 'hypercall'. QEMU's binary translator allows an implementation of such a mechanism.

QEMU distinguishes between simple operations, which can be directly converted to micro operations and complex operations, which need further processing with host code. The latter is realized with helper functions. A helper function is called every time, a complex or privileged instruction is encountered, which cannot be translated to simple micro instructions. We utilize these helper functions and their architectural specific handler routines to augment them with a call to our specific hook function. The general process is depicted in Figure 5.5.

Every instrumented Linux function collects the required data, e.g., the process identifier and its corresponding page table directory for a process creation event, copies the data into a buffer located on the current kernel stack and issues a hypercall, which calls directly into the simulator.

Hypercall Interface

We focused our implementation on IA32, compatible to x86 and its 64 bit extension. IA32 contains *model specific registers* (MSRs) to configure CPU states, enable power-saving features, and to retrieve internal CPU details. It provides instructions (`rdmsr` and `wrmsr`) to get and set these registers via a register address. The IA32 specification declares an unused register address range (0x400000000-0x400000FF), which will remain unused in the future [24]. Using these instructions with an invalid register address on a real system will cause an exception, terminating the running process or the whole OS. As QEMU handles them with a specific helper function, we intercepted this mechanism and call our OS intro-

Purpose	Data to collect
Memory Allocations	• Kernel Memory Allocations • Buddy Allocations • Page Cache Allocations
Scheduling	• Dispatch Events
Task	• Fork Events • Exec Events
Address Space	• Creation, Destruction • VMA Allocations • Memory Mappings • Page Table Manipulations
Memory Layout	• Static Kernel Layout • Dynamic Kernel Layout • Module Loads

Table 5.2: Recorded Linux system events. They enable a reconstruction of the current system state and enrich recorded memory accesses with semantic information.

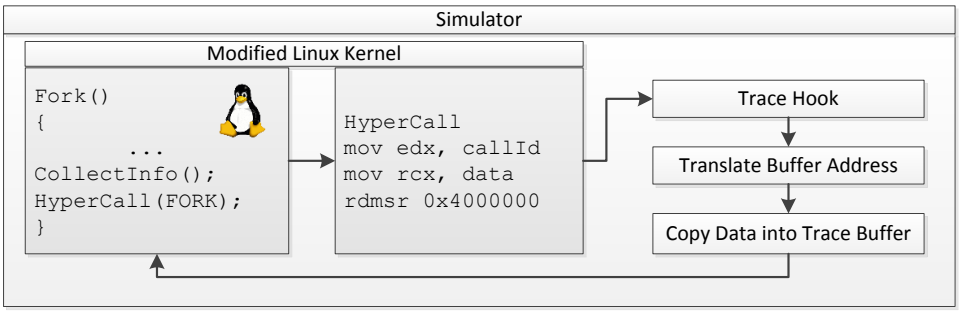

Figure 5.5: An instrumented Linux kernel collects necessary information and issues a hypercall, which calls directly into the simulation. The simulator retrieves the contained information and stores it into associated data streams.

spection hook instead. The hook receives a pointer to a data buffer and a hypercall ID via general purpose registers. As we try to minimize the impact on the simulated OS, we chose the `rdmsr` instruction as the magical instruction, because a `wrmsr` might cause a CPU state change. QEMU calls the hook, when a code path reaches a `rdmsr` instruction with our hypercall address (0x40000000) and redirects it to the hook. Any processing of the contained information must be done in a secondary step. The hook translates the pointer to our data structure on the Linux kernel stack in guest memory to a host accessible address. It must further resolve the pointers within the transferred structures before the trace hook can access the referenced data. This is necessary, because the returned pointers are only valid within the guest kernel's address space. To speed-up the processing only few instrumented functions return strings, e.g., the `exec()` system call contains image name, command line parameters, and the execution environment as pointers to strings. Other data structures, e.g., page table modifications, can directly be copied without a secondary address translation.

The current introspection is only compatible with Linux and a modified QEMU version, since other VMs and simulators do not ignore the invalid register value chosen for our hypercall and stop further code execution. A more sophisticated approach could use breakpoints based on symbol information. Such a solution, albeit much more complex, might also be capable of introspecting a closed-source OS. However, this was out of the scope of this work.

5.2 Storage Server

Previous implementations of the data acquisition framework showed limitations in portability, compatibility and scalability. The original storage component of Rittinghaus [41] consists of a simulator (Simics) and data analysis framework both running on Windows. In the following we describe our implementation changes to increase portability, compatibility and scalability.

As we use QEMU for data acquisition and its support for Windows is limited, we extended the previous design and implemented a storage server running on POSIX-compatible operating system as well as on Windows. To retain compatibility to different simulators we introduced a small library which serves as front-end for the storage server.

5.2.1 Storage Server Interface

The storage server provides a light-weight interface to its functionality. The interface is encapsulated in a library to ease integration into different clients. Figure 5.6 shows the integration of the library into a simulator and its interaction with the storage server.

Our library handles connections, stream registration, and hides details of trace buffer management from both simulation and analysis framework. Once a client connects to a server instance, the library creates a dedicated session with private shared memory buffers. These buffers are mapped into the address space of the client and are used for trace data exchange. A simulation, for instance just emits trace data into these stream buffers. If a buffer exceeds its capacity, it is transparently replaced by the storage server with a newly assigned buffer, while the full buffer is asynchronously compressed and written to disk.

Furthermore, the library provides enumeration functionality for stored traces, its contained streams, and associated meta-data. To open a trace file, the analysis framework establishes a connection to the storage server and opens a data store by name. Then the server will transparently parse the trace data file and present the contents of the trace to the client. Instead of registering a new stream, as a simulation would do, the client opens saved streams and requests meta-information, concerning the size, the entry count, and the stream type. The stream data can then be accessed on a per-entry basis or as a complete segment. In both cases the library maps the data directly into the analysis framework's address space and presents its content as a local array, or depending on the client's programming language as an enumerable resource.

As described in Chapter 4, the previous file format is a complete stream oriented storage with variable-sized segments and trace entries. We keep the idea of

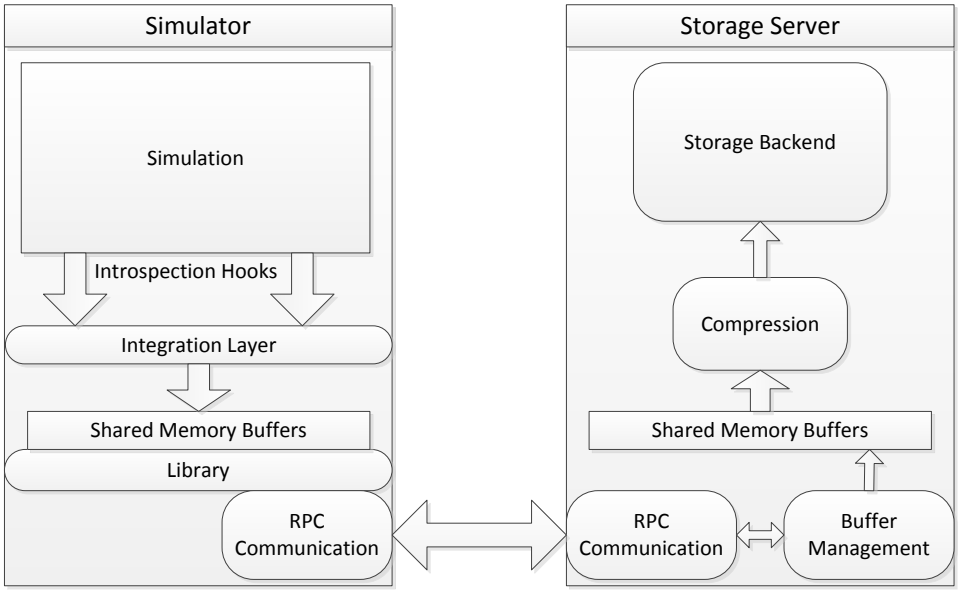

Figure 5.6: The different abstraction layers, which provide a simple integration of the storage server into various simulators. All interfaces are exposed through a small library.

streams, but change the general data layout to allow indexing and fast access to all segments associated with a data stream. To achieve fast indexing the size of each segment and trace entry must be of fixed size.

5.2.2 Revised Trace File Format

Although the storage server is not bound to a single storage back-end, recent versions implement a file based storage approach based on the original trace format of [41]. As the primary problem of the original framework is scalability, we had to improve the trace file format to allow fast opening and data retrieval for large traces. Our file format comprises three distinct objects as illustrated in Figure 5.7: A descriptive *file header*, containing meta-data for the complete trace file, *directories*, which allow fast access to each data stream and *stream segments*.

The file header provides statistics, file integrity check information, and direct access to all directories. The statistical data include the count of streams, the amount of trace entries, the compressed and uncompressed file sizes, for fast integrity checks, and time information describing the recorded simulation. This header is directly followed by an initially allocated empty directory. A *directory*,

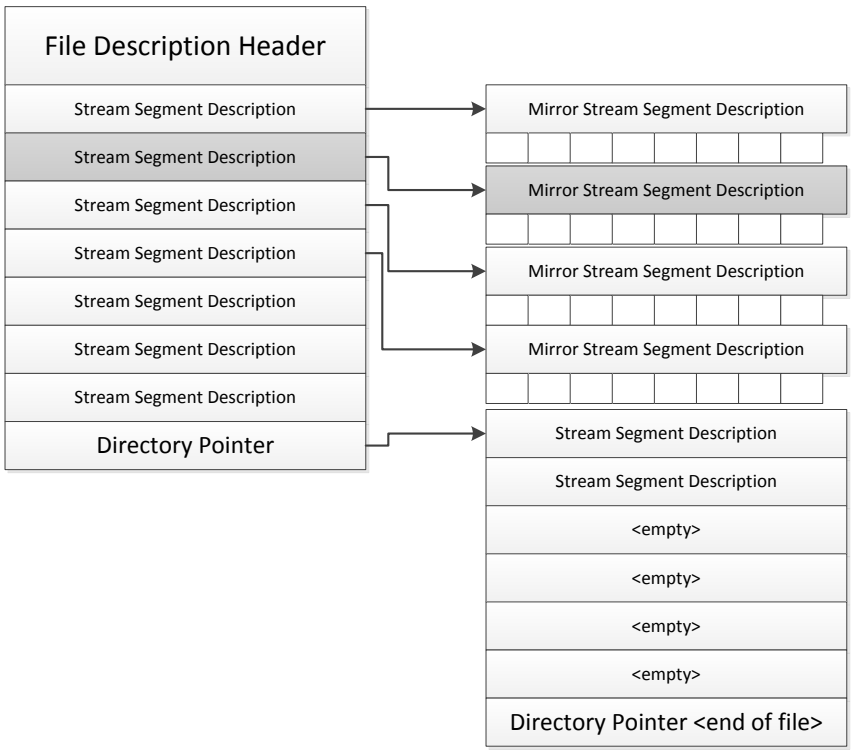

Figure 5.7: The layout of a trace file. It consists of a file header for metadata, directories, for fast stream access, and stream segments, for actual data storage.

in this file format, is similar to a regular file-system directory. It contains metadata describing referenced fixed-sized *stream segments* and a pointer to their actual storage location. The directory structure makes it possible that the complete file content can be loaded instantly. Discovering streams and their associated segments is by magnitudes faster than in the previous sequential file format. Accessing a segment and its trace entries, requires an initial lookup in the typically buffered directory structure.

Trace Entry In the previous file format, trace entries are compositions of one or more variant data blocks. A data block contains 10 bytes, one describing its content and 9 for actual data storage. Larger trace entries are a composition of these data blocks. To overcome the slow scanning before stored data can be accessed, we enforce that a stream contains *only one* type of trace entry to improve the *data accessibility* and allow simple indexing.

A trace entry is the smallest unit of data. It contains elementary data types such as integers, bit fields, and strings. Every trace entry contains at least a timestamp to identify the temporal correlation within other data streams. Typical trace entries used for our data collection are depicted in Figure 5.8. These structures are passed to the storage server and retrieved by clients.

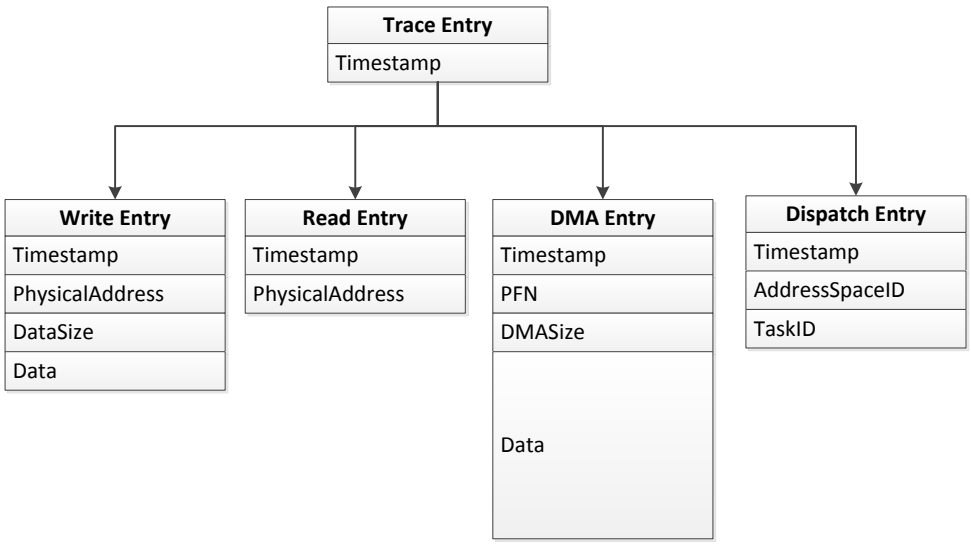

Figure 5.8: Typical trace entries, as actually used for the data acquisition. Every entry consists of a timestamp to correlate it with other data streams.

Although the trace entry size is fixed for each stream, variable data, e.g., strings can still be stored. As strings for OS introspection typically have different length, e.g., filenames to identify page content or process environments, storing a fixed-size char array in every entry is unfeasible. Therefore, a string is broken down into small data portions and stored in an associated *string pool*.

The String Pool Strings are all pooled in a special data stream and not directly stored in a trace entry. That enables all trace entries to contain strings, or other variable sized data and still retain their fixed-size and their simple indexing scheme. Strings are broken down into fixed-size data blocks as seen in Figure 5.9. Each data block comprises a control sequence of one byte, followed by the actual string data (15 bytes). The index to the first block is returned as a *unique string id* and stored in the trace entry instead. All following items belong to the same string, till a new and therefore terminating control block is discovered. A reason to pool string data is the possibility to deduplicate them on-the-fly and as strings

contain a limited set of characters, the probability to achieve better compression ratios is higher, as if scattered across different streams.

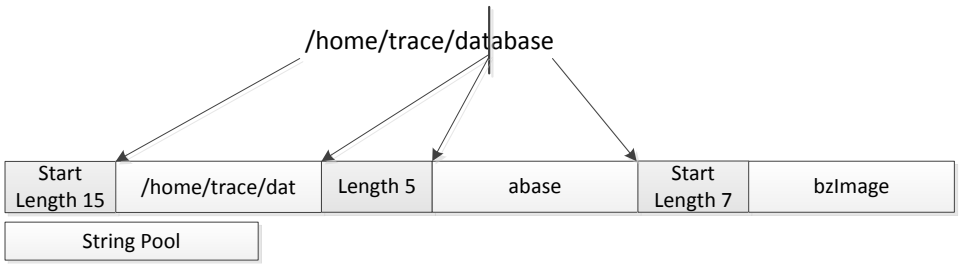

Figure 5.9: Every string is broken down into fixed-sized data blocks. Each string is pooled to deduplicate redundant data.

Compression To cope with large amounts of data and limited backing store, trace data is compressed. In the original implementation with a strong, but slow compression algorithm (LZMA). Although, simulation with Simics is comparably slow, there were still problems to process and compress the resulting amount of trace data [41]. The quantity of generated data and the slow compression interrupted the simulation, every time the buffer inside the simulator had been exhausted. As we expect even more data from QEMU in shorter time intervals, we chose to replace the compression algorithm with a fast version of deflate. This version has only a slightly worse compression ratio, but is faster than LZMA [16]. Early tests with QEMU confirmed, that we can expect 24 MIPS for the guest and about 4 million write operations per second (about 100 MiB/s uncompressed data). To further improve compression of timestamps and addresses we augmented deflate with delta-compression.

As the storage server transparently compresses and decompresses trace data with different algorithms, the simulation can chose from three different compression implementations: a *null compressor*, which leaves the trace data unchanged, a *deflate compressor*, e.g., for the string pool, and a *delta-enhanced deflate* algorithm.

For example, a write operation, typically writes to adjacent addresses, which are ideal candidates for delta-compression. As every trace entry contains a timestamp, frequent operations can benefit from delta-compression as well. For rare events a delta-compression, however, can decrease the compression ratio of the deflate algorithm. In our evaluation, we observed compression ratios to vary between 1:4 for writes with random data, and 1:400 for memory zeroing operations.

5.2.3 Data Recording

A typical data recording scenario is depicted in Figure 5.10. The simulator, in our case QEMU, uses the provided interfaces exposed by our library to connect to an instance of the storage server and register different data streams. The minimal amount of streams for our analyses contain memory store streams, containing all memory modifications, a DMA stream, and all OS introspection streams[2]. Once the streams are registered, the storage server allocates a suitable amount of shared memory to provide backing store for stream segments and map these regions to the simulation's address space. The handling of different address space sizes is transparently done by the storage server. As the simulation progresses, it fills the provided buffers with trace entries, till its capacity limit is reached. The API layer handles that transparently for the simulator. It signalizes that a segment can be asynchronously processed in the storage server and returns a new buffer for this stream. While the storage server compresses and stores the segment to disk, the processing can continue without interrupting the simulation process. If the simulation has finished its work, it releases all registered streams, and closes the connection. The storage server will then process all outstanding requests, finalize the storage container, and release allocated resources. Only a small integration layer is necessary to allow inspection hooks to produce data.

5.3 Trace Data Processing

The storage server offers interfaces for storing and retrieving data. Once a simulation has produced trace data, an analysis tool can access them through an instance of the storage server. For this purpose it utilizes our library.

Once a simulation has finished, the analysis can work with the created trace data. Although it is possible to inspect the data during recording, a full reconstruction of the system state should be postponed, to reduce the impact on the backing store's bandwidth and the simulations' progress. The client, in our case a slightly modified version of the *trace viewer*[3] [41], connects to the *storage server*, opens a store, and accesses all required trace data streams. Since storage, compression, and further processing is done transparently, the client can access all trace data entries, as if they were contained in a local array in its own address space. If it reaches the end of the supplied buffer, it can request the next stream segment through the small API layer, which encapsulates buffer and meta-data management.

[2]Please remember that each stream may contain only one type of trace entry (i.e., information). For that reason, multiple streams are allocated.

[3]The trace viewer is a GUI application and an interface to the analysis framework.

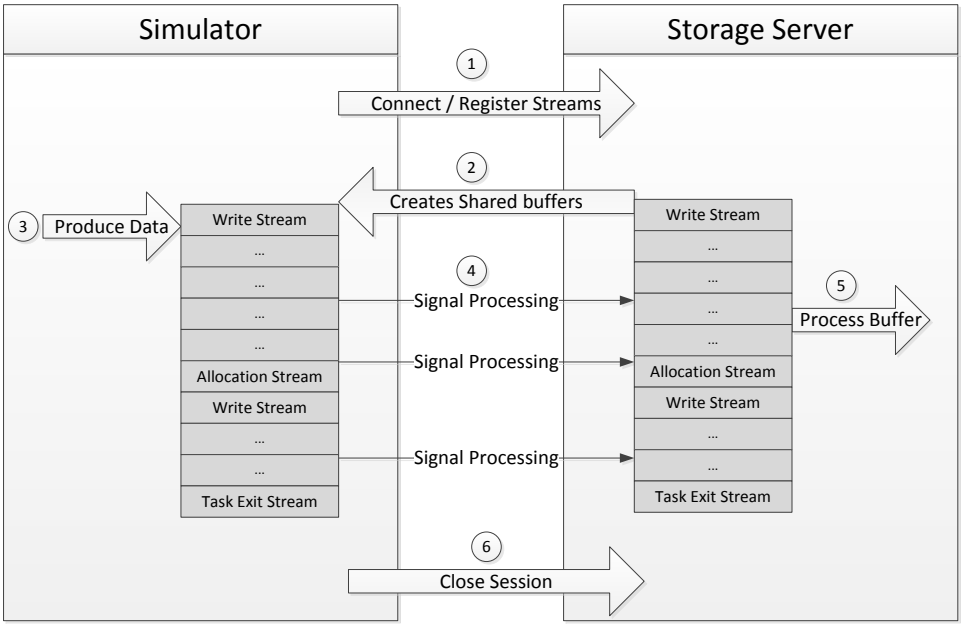

Figure 5.10: A typical service scenario: A simulator connects to an instance of a storage server and registers its streams. In return a shared buffer is created and the simulation can produce and commit trace data.

We fed trace data into two different reconstruction units, the *memory reconstruction* and the *OS state recovery*. After these two steps, we combine their results into a single *frame history* to reduce the memory requirements of the analysis framework.

Memory Reconstruction

The memory reconstruction process uses the recorded memory accesses, to reconstruct the original memory content with instruction count accuracy. As illustrated in Figure 5.11, the different streams – CPU memory writes and DMA writes – are multiplexed according to their temporal correlation. The memory reconstruction recreates the system RAM of the simulated system by replaying the write and DMA operations on an equally sized memory array. After each reconstruction step, the memory content is identical to the simulated guest memory as it was at the same point in time of the simulation. If two different recorded traces should be correlated to each other, that is done in parallel, working with a single instance of the storage server. The limiting factor is only the available hardware resources. To

obtain *sharing opportunities*, the analysis component invokes a sharing analysis after the memory image for a specified timestamp is ready.

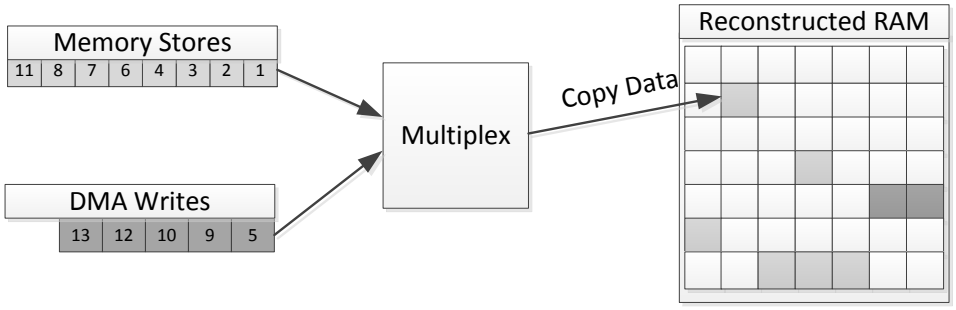

Figure 5.11: The memory reconstitution process multiplexes different memory streams to create a memory image for the specified timestamp.

Sharing Analysis

The main problem of discovering sharing opportunities is its exhausting and time consuming nature, since every page's content must be compared to every other page's content. That leads to a complexity of $O(n^2)$. As previous work suggested [41, 52], this task can be simplified by hashing every page's content and using this digest as an index into a lookup structure. This can reduce the complexity to $O(n)$. Depending on the hash function, the likeliness for *false positives* – wrongly identified sharing opportunities – is too high for reliable analyses. Therefore, a full page's content comparison is needed. The approach taken by our analysis is depicted in Figure 5.12. For the first part of the sharing analysis, we utilize a *hash map* with chained entries. The index into the hash map is created with *cyclic redundancy check* (CRC). As a CRC32 is fast, but not very reliable and many hash collision exists, a hit into the hash map does not imply that an identical page frame has been found yet. Only after a successful byte-wise comparison of each referenced frames' content, the page frames are considered identical. This methodology is applied to both intra- and inter-domain sharing analyses. In the case of the *intra-domain* sharing analysis, every page is hashed and inserted into the hash map. A collision with a successful comparison marks each frame as shareable within its own domain. For *inter-domain* sharings a second step follows as illustrated in Figure 5.13. After an intra-domain analysis, each hash map represents the memory content of each trace up to this point in time. The inter-domain sharing analysis relies on two or more reconstructed memory images and

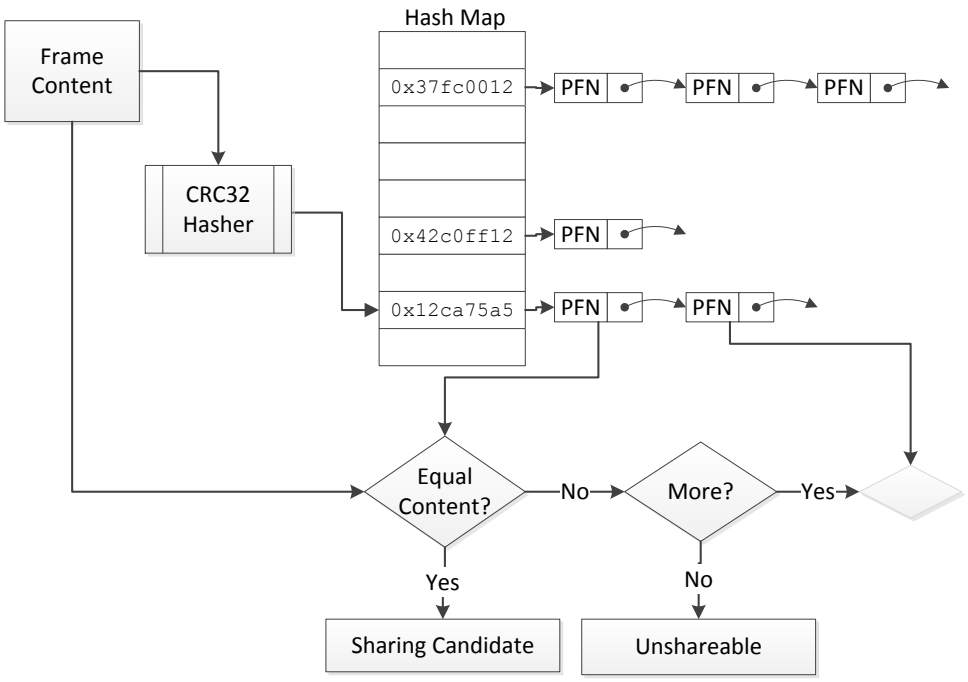

Figure 5.12: The process from page frame content to the detection of a sharing opportunities. The chained hash map speeds up detection and avoids false positives.

their hashed representation. For every entry in a hash map, all other hash maps are queried, whether they contain an identical hash or not. If a hash is found, all pages associated with this hash map entry are compared to the matching entry's linked list. So every possibly page frame content is compared with all possible changed page contents in the other hash map. On success, the frames are marked as inter-domain shareable. This allows a fast and precise sharing analysis, without the risk of missing sharing opportunities or producing *false positives*, due to hash collisions.

Once the page frame statistics have been collected, they still lack semantic information, i.e., their current usage inside the guest Linux. To enrich page frames with allocation information, the OS state is reconstructed in parallel and fused with the previously reconstruction information and the generated statistics.

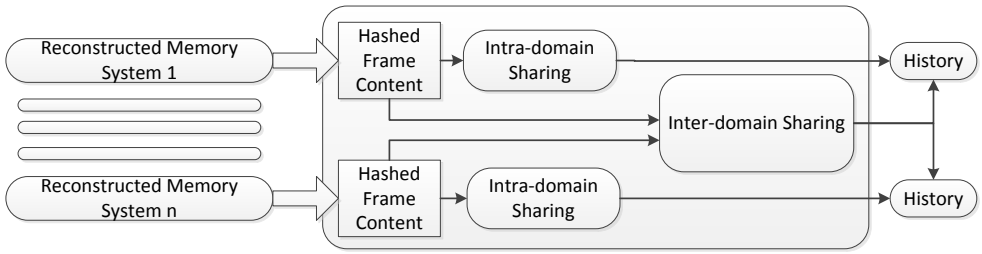

Figure 5.13: To find possible sharing opportunities, every memory snapshot is fed to a hash map. These hash maps are compared to retrieve all inter-domain shareable pages.

OS State Recovery Along the memory reconstruction, the semantic information must be extracted and associated with each page frame. Each trace entry contains a timestamp to correlate allocation events with a specific memory state. This state is reconstructed and multiplexed with the memory information. The process is similar to the original implementation, only extended to comply with a 64 bit environment. This reconstruction can be done in two ways.

In a *simplified reconstruction mode*, only a sub-set of OS introspection streams are processed. It only considers buddy allocation events, page cache allocations, and address space page frame mapping streams. This adds the missing semantics back to each page frame.

Another and more complex way is an *advanced reconstruction mode*. It reconstructs not only memory semantics, but also complete address spaces, their associated tasks and its internal layout. Furthermore, it replays the process dispatching with the recorded scheduling information. That allows to correlate each write operation to a process and corresponding thread.

While the results in our evaluation did not integrate the information available in the advanced reconstruction, we frequently used the data to facilitate our understanding of a system's behavior.

State Compactification

The main problem of the reconstruction processes is the vast amount of required memory. Keeping a history of previous states, rapidly exhausts all system resources, even for short time periods. As the main purpose of the reconstructed memory image is to allow a false positive free sharing analysis, the memory content can be discarded after successfully extracting sharing information and statistics collected during reconstruction.

To reduce the required memory even further, we use *temporal sampling*. For this purpose the system state is reconstructed in *fixed time-intervals* or *event driven*, depending on the specific question. Once a memory and system state reconstruction have reached the specified simulation time, the sharing analysis takes place and other information, such as write count, sharing- , and semantic information are extracted from the system state and combined to a single representation. The time-interval for sampling has to be chosen carefully. On the one hand, a large interval ignores short lived sharing opportunities as previous research did [36]. On the other hand, a too small interval shows sharing opportunities that can hardly be considered for sharing, because they only exist for such short periods (e.g., 20 ms) that there is no justifiable benefit from merging them. The optimal interval depends on the analyzed problem. To show the potential for very short lived sharings, a sub second granularity is advantageous, whereas for scanner improvements a resolution of one second is already sufficient. A well-chosen sampling rate allows keeping the complete history of each page frame within reasonable storage space. For instance a complete trace file occupying 176 GiB and containing approximately 55 billion memory modification events, can be reduced to a 3 GiB frame history file. If the time-interval does not provide the required information or a different question should be evaluated, the trace data can be sampled again, without requiring a new simulation.

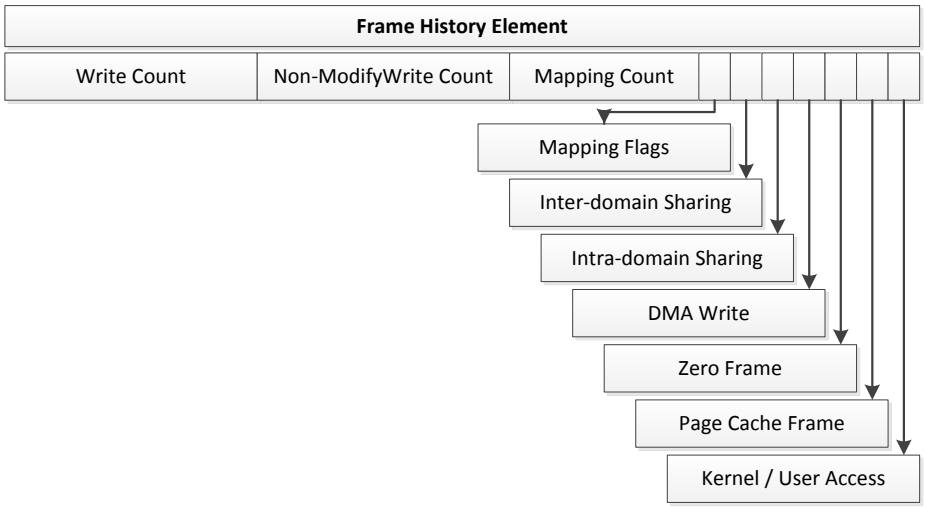

Figure 5.14: A frame history element. It contains access information, mapping count, page usage, and sharing information for a page frame.

We implemented the history of each page frame with bit fields and counters (both 32 bit) to have a compact representation, counting the occurred write ac-

cesses, non-modifying writes, and mappings of each page frame. All counter overflows are reported during creation and probably indicate a too large sampling interval. A *frame history element* is depicted in Figure 5.14. The bit field contains the allocation, e.g., if a frame is used as *named or anonymous memory*, sharing information, e.g., *inter- or intra-domain*, whether it was subject to *DMA operations*, and how the page was written recently, e.g., by *kernel or user space*. For each snapshot of the reconstructed system and for every page frame such entry is saved to disk for further analysis.

To reduce the consumed memory on disk and during processing, only *accessed* frames occur in a *frame history file*. A frame history file containing typical access sequences with semantic information can be seen in abstract form in Figure 5.15. The unused, but probably allocated guest physical pages can be ignored, since the guest system considers these pages as unused or zero. That is different from page frames marked as zero, since these frames have already been used and were actively zeroed or released.

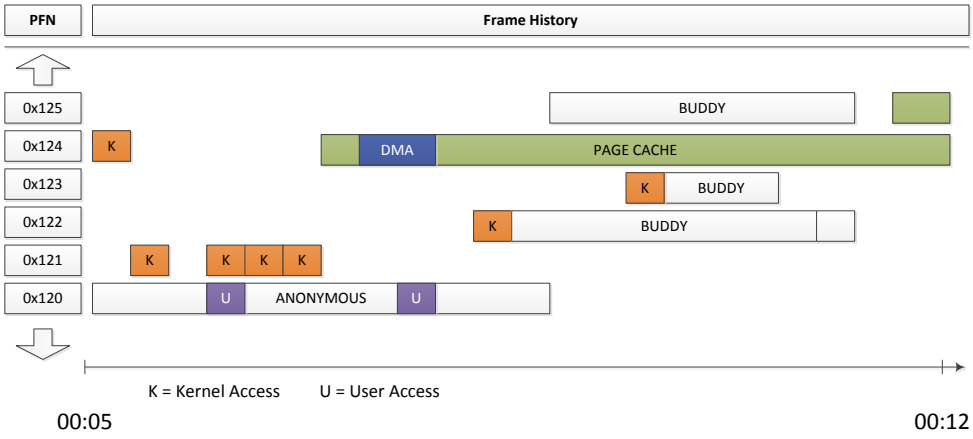

Figure 5.15: A typical frame history file layout. Showing the temporal development of each touched page frame. Different events are correlated into a single compact view.

5.4 Conclusion

For our implementation, we modified the analysis tool set of previous work, implemented a revised storage component, and extended QEMU.

We extended QEMU to comply to our data acquisition needs, to collect memory accesses, added a reliable time source to allow data correlation within a simu-

lation and augmented it with a hypercall interface to collect OS introspection data. To acquire semantic information from our chosen guest OS, we instrumented a recent Linux kernel to transport the internal system events to the simulator. All memory modifications and semantic data is stored in buffers supplied and managed by the storage server and its client library. Our revised storage server acts as a transparent data bridge between simulator and analysis framework, hiding storage and compression details, while providing fast and random data access under Linux and Windows. To increase scalability and avoid cumbersome analyses we extended the trace file format with directories, which allow faster and direct access to stored trace data.

Our sharing analysis has moved to the framework and works on reconstructed memory images, allowing false positive free sharing opportunity analyses. As we record memory modifications instead of hashes, it further allows spatial and temporal sampling on a single recorded simulation trace. In conjunction with the slightly modified OS state reconstruction, we can analyze sharing opportunities, memory access frequencies, and access patterns enriched with semantic information. To achieve better performance and keep a write access history of every page frame, we developed a secondary step that compactifies and optionally samples the resulting analysis information into a frame history representation. That allows more complex analyses without the need to repeatedly reconstruct memory images for each analysis pass.

Chapter 6

Evaluation

In the previous chapter, we proposed a data acquisition system composed of a full-system simulator, data storage component, and a memory reconstruction and analysis framework. QEMU serves as full-system simulator and creates trace data which is analyzed with our proposed tool set.

This chapter presents our analysis methodology, evaluation setup and results of a sub-set of evaluated benchmarks.

6.1 Methodology

With this evaluation, we try to find correlations between stability and memory access frequencies as well as memory access patterns. We further want to investigate correlations between memory semantics, sharing opportunities and stability, as there is sharing potential not or slowly harvest through scanning. We aim to find hints to focus a scanner on suitable pages or exclude unstable pages temporarily or permanently from memory scanning.

To achieve that we evaluate a wide-range of different benchmarks that range from I/O intensive to CPU- and memory bound real-world workloads. With these different workloads it should be possible to cover common cases and identify stability and sharability criteria, if they exist.

We start with the evaluation of sharing opportunities to identify the overall sharing potential and to correlate our results with other work. Afterwards, we investigate memory access frequencies and access patterns to infer stability from outside an application or VM. Further, we focus on page stability in conjunction with semantic information from our guest OS introspection to analyze the remaining sharing potential, and if harvesting such potentials is worth the afford. We end with an examination of write working sets as they might help to reduce scan overhead by including stable and excluding unstable content.

Our core analysis utility is a *Hidden Markov Model* (HMM) to determine *stability* associated with semantic information. A HMM allows modeling a state with hidden properties. Each state in a HMM depends only on the previous state, the past of each state – its history – is ignored. Our HMM state graph contains typical page frame states, such as allocation and stability.

Previous research on the field of file content deduplication suggested a sophisticated state model, which evaluates the probabilities of state transitions between different file states [49]. We adapted a similar model, but for memory content. Such model allows analyzing the stability of *page frames* during benchmark execution with memory allocation information. We therefore created a HMM and modeled memory transitions. A memory transition is a change from one page frame state to another caused by: (1) a memory store operation, (2) a DMA access, (3) an unmapping or remapping event, or (4) with progression of time. Our state model consists of five different memory usage scenarios. A page frame can either be allocated through the buddy allocator of Linux [7] to the *page cache*, directly used by the OS, a so-called *buddy page*, or it can further be mapped as *anonymous memory* to a user land process, e.g., for a process heap or uninitialized data of an executable file. The other two categories consists of memory assigned to kernel stacks, device memory, and slab allocators, referred to as *other* or a currently *freed* frame, originally allocated for one of the four other categories.

Categories	Short Description
Page Cache	File related page frames
Anonymous Pages	Process heaps, uninitialized data
Buddy Pages	Virtual kernel allocations
Other Pages	Kernel stacks, device I/O memory, slab allocation
Free and File	Freed and not reused page cache pages
Free and Anonymous	Released anonymous memory
Transitions	
DMA Write	DMA operation directed to page frame
Memory Write	Memory modification to page frame
Idle Time	Time in which no modification of the state occurs
Mapping	Allocations, re-mapping, release of page frames

Table 6.1: Summary of all categories and transitions modeled for our HMM.

The resulting transition probabilities can be used to predict the future development of every page frame that reached the same state, and is therefore well-suited to understand *page stability*.

We distinguish between several page states. The most important states, considering page deduplication, are *stable states*. These states are reached, when no write access or mapping event occurs for a certain predefined amount of time, e.g., for more than 2 seconds. An optimal stable criterion depends on the aim of the deduplication system. Should either more page frames become available and thus allow more workloads to run in parallel, or should only long lasting content be deduplicated with less computational overhead. We chose for our experiments different stability criteria, ranging from 1 second up to 4 minutes.

All other states are considered *unstable*. When a write operation occurs in a *stable state*, the state transits back to an *unstable state*. In the examined OS, this transition would affect an already merged page and trigger the copy-on-write mechanism, which ensures address space isolation by creating a private copy. Such transitions will increase the memory footprint of a VM. A simplified illustration of our HMM can be seen in Figure 6.1. We distinct between content that has originated from backing store and becomes stable – *stable page cache* – and pages that have been modified or created – *stable written page*.

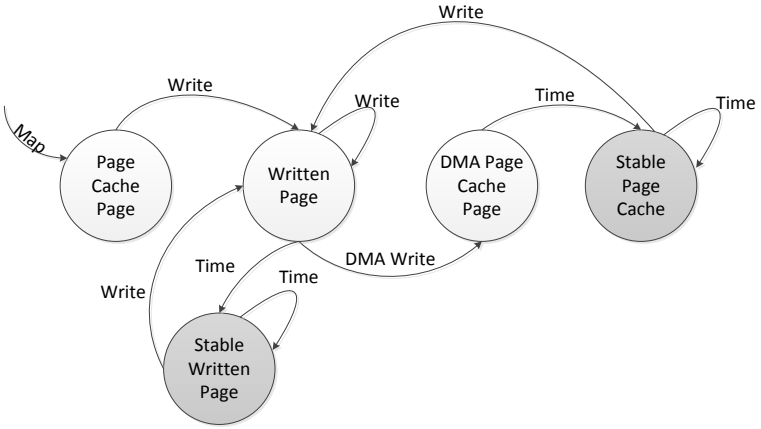

Figure 6.1: A simplified HMM for the Linux page cache. It shows a sub-set of possible transitions. All unmap transitions are omitted.

With all modeled page categories and transitions found in Table 6.1, our complete HMM state graph consists of 71 nodes and 276 edges. It allows a thorough analysis of page frame *stability* and can be correlated to *sharing opportunities*.

6.2 Evaluation Setup

For our evaluation we use the proposed implementation from Chapter 5 to acquire trace data for different workloads. We use different benchmarks to cover a wide range of different memory usage patterns, including excessive use of file caches and anonymous memory.

6.2.1 System Configuration

As we utilize QEMU, a full-system simulator for data acquisition, our evaluation setup consists of two different systems: a physical host executing the simulator and handling the data transfer to our network attached backing store and the simulated system running in our adapted QEMU environment.

Host System Configuration The host system contains 2 multi-core Intel Xeon processors with 12 physical, and 24 logical cores. Its computational power allows to run all benchmarks without interrupting or stalling the simulation. During testing we discovered that an old Intel QuadCore Q9450 was not able to achieve maximal performance, as the single-thread performance was too low. On average 4 cores were solely dedicated to compress incoming trace data streams and store the compressed data segments to our network storage. As QEMU only supports single-threaded simulation, even when simulating a multi-core architecture, the maximal achievable amount of traceable data is limited. Our storage server had to compress about 128 MiB of raw trace data per second. The complete host configuration can be found in Table 6.2.

Component	Model / Specification
CPUs	2x Intel Xeon CPU E5-2420
Available Cores	12 (24 logical cores)
Frequency	1.9 GHz to 2.4 GHz
Memory	12 GiB (ECC DDR3-1333)
Attached Storage	8 TiB Network-attached RAID Storage
Operating System	Ubuntu Server 12.04
Kernel Version	Ubuntu 3.2.0-51-generic
Architecture	64 Bit (amd64)
Simulator	QEMU Version 1.3.1
Simulator Extension	Memory Hooks, Hypercall Interface, Instruction Counter

Table 6.2: The configuration of our host running QEMU.

Simulation Configuration All benchmarks were executed inside our modified QEMU version with activated *icounter* (for full details see Chapter 5). The icount mode allows an undisturbed workload execution, hiding timing distortions due to introspection and tracing overhead from the guest operating system. Although this is not true for our configuration of the instruction counter, some, only slightly different configurations, lead to an unresponsive binary translator and to stalled virtual CPUs. Unfortunately, this happens on all recent versions of QEMU running binary translated code on x86 processors, with enabled icount. Some Linux version seem to further affect this behavior, whereas a 32 bit version runs without problems, a 64 bit version experiences different timings and either stops working – is unresponsive – or initiates a *kernel panic* in the guest OS. We chose to simulate a 1 GHz CPU, since that is the only configuration which runs correctly. Further QEMU problems and possible improvements can be found in Future Work 7.1.

Our simulated system is typically equipped with 1 GiB of RAM. That allows an execution of all benchmarks without memory pressure. If memory pressure is desired, we will state the configuration changes. All evaluated workloads run on the same guest OS. It is a minimal Ubuntu Server 13.04 installation with a customized 64 bit kernel. The vanilla Linux kernel version 3.9.2 is augmented with hypercall extensions for OS introspection as described in Chapter 5. No additional background daemons have been added, but we deactivated some daemons, e.g., the firewall and KSM, to focus on the execution of the examined workloads. A complete list of all configuration details can be found in Table 6.3.

Component	Model / Specification
CPU	simulated 1 GHz
Memory	1 GiB guest memory
VGA	Cirrus Logic GD 5446 VGA emulation
Network	Not connected
Storage	32 GiB (QEMU Enhanced Disk Image)
Guest OS	Ubuntu Server 13.04
Kernel Version	Linux Vanilla Kernel 3.9.2 with OS Introspection
Architecture	64 bit (amd64)
Simulated Architecture	x86-64
Simulated CPUs	1
Simulation Mode	softMMU, TCG, icounter, optimized build

Table 6.3: Simulator and guest OS configuration used to execute our workloads.

6.2.2 Benchmark Setup

We performed different benchmarks to collect trace data to identify new hinting sources, analyze access patterns and sharing opportunities. In the following, we describe the characteristics of each benchmark. If not noted otherwise all benchmarks are unmodified 64 bit versions, as found in the Ubuntu repository.

Linux Kernel Build The compilation of the Linux kernel is a common benchmark and often used to show the effectiveness of memory deduplication mechanisms [11, 35, 36]. Therefore it is obligatory to examine this benchmark as well. We compile the vanilla Linux kernel version 3.10.0 with *two instances* of a C compiler (gcc) executing in parallel and afterwards emit a compressed Linux kernel image. This benchmark completes in about 15 minutes on real hardware. In a QEMU simulation with introspection it takes about 4 hours and 15 minutes, which is still faster than other full-system simulators. The benchmark fills the page cache, with source and output files, and tends to retain all required data in the page cache, when enough RAM – about 320 MiB – is available. We executed this benchmark in two different configurations, with 256 MiB and 1024 MiB simulated system RAM. Memory deduplication utilizing hints based on I/O operations can deduplicate large parts of this workload. There are two reasons for this behavior. Firstly, this benchmark does only change few files on disk, thus a file can remain in the page cache for long periods of time. Secondly, most created files remain unchanged once they are written to disk.

SPEC Benchmark Suite The SPEC benchmarks of the CPU2006 suite provide different real world scenarios to measure and compare different computer systems [47]. These benchmarks range from simple compression tasks to complex quantum mechanical simulations.

In contrast to the kernel build, they run completely in memory. Even benchmarks such as *403.gcc*, which compiles and optimizes program code, do not write any resulting data to disk. In the case of the gcc benchmark, the required source-code files are loaded during the start-up phase and no further I/O operations to disk occur during a run. We have executed different benchmarks to simulate their behavior of a CPU- and memory-bound workload. All evaluated benchmarks remained unmodified, as found on the installation medium supplied by SPEC. We chose three distinct benchmarks, *403.gcc* and *464.h264ref* from the CINT2006 suite and *433.milc* from CFP2006. These are typical workloads, where I/O hinting cannot achieve high sharing rates, since only the initially loaded benchmark data and OS files can be deduplicated. As the generated output resides in main memory, they can only be deduplicated by scanning.

Benchmark	Application Area	Brief Description
CINT2006		
403.gcc	C Compiler	Based on gcc Version 3.2, generates code for Opteron.
464.h264ref	Video Compression	A reference implementation of H.264/AVC, encodes a videostream using 2 parameter sets. The H.264/AVC standard is expected to replace MPEG2
CFP2006		
433.milc	Physics / Quantum Chromodynamics	A gauge field generating program for lattice gauge theory programs with dynamical quarks.

Table 6.4: The chosen benchmarks from SPEC CPU2006. The benchmark descriptions are taken from SPEC.org [47].

Phoronix Benchmark Suite Phoronix Benchmark Suite provides different, every day workloads [34], typically to measure the performance of Linux and UNIX based systems. It provides five benchmark categories to measure different performance aspects: System, Disk, CPU, Graphics, and Memory. We chose different benchmarks focusing on the overall system performance and disk utilization, since those two categories are not covered by SPEC CPU2006.

Benchmark	Brief Description
Phoronix Database Suite	
pts/sqlite	Measures the time to perform a pre-defined number of insertions on an indexed database.
pts/pgbench	TPC-B like benchmark of PostgreSQL.

Table 6.5: Chosen tests from Phoronix Database Suite. The benchmark description is taken from openbenchmarking.org [33].

Some benchmarks caused problems and could not be used for our evaluation. It is not completely clear why they behave different in a virtualized (e.g., VirtualBox) or simulated system (e.g., QEMU), but in both cases they were unable to complete their tests successfully. The benchmark based on the openJDK Java implementation crashes during tests and seams not to work in recent versions of

QEMU, at all. Therefore, we limited our investigation to the database benchmark suite, which consists of two benchmarks that perform real-world database transactions in with SQL [33]. A more detailed description of the benchmarks can be found in Table 6.5.

Bonnie++ This benchmark measures hard disk performance [13], it stresses the page cache as well as the file system implementation. In our case the latter is unimportant, we are only interested in its behavior towards file caches and the typically resulting sharing opportunities and their lifetime. As its workload is completely different from other evaluated benchmarks, it produces a unique page cache sharing pattern. As pages allocated to the page cache remain stable for long periods of time and show large amounts of sharing opportunities [36, 41], this benchmark, on the contrary, rapidly changes file content and forces the page cache to drop page frames on a regular basis.

In the following, we refer to these five chosen benchmarks with abbreviations found in Table 6.6. They cover nearly all found behaviors of our examined workloads.

Abbreviation	Benchmark	RAM Size
KB1024	Linux Kernel Build	1024 MiB
KB256	Linux Kernel Build	256 MiB
B++	Bonnie++ Benchmark	1024 MiB
PDB	Phoronix Database Benchmark	1024 MiB
GCC	SPEC 403.gcc Benchmark	1024 MiB

Table 6.6: Abbreviations for selected benchmarks.

6.3 Benchmark Evaluation

Since our benchmarks run inside a simulated system and are temporally sampled, time information cannot be taken with a wall-clock. As our internal time is measured with an instruction counter, a time approximation is necessary to make our results comparable with real systems. In the following sections we describe how we solved this issue. We further give details about the handling of zero page frames in our statistics, because previous studies disagree in this point.

6.3.1 Simulation Time Estimation

The measurement of time inside a simulated system is difficult. As described earlier, there is no reliable way to measure the simulation time with the required accuracy from within a simulated guest in QEMU. Our instruction counter enables us to count all executed guest instructions. Although such time measurement ensures a correlation of data and allows to compare progress, speed, and further judge the amount of work done, it does not allow a simple correlation with real time. There are at least two correlation problems.

Firstly, on a real system every instruction takes a different amount of cycles to complete, e.g., a simple `add` instruction might consume a single cycle, whereas a division (`idiv`) can consume up to 87 cycles [15]. The actual cycle count heavily depends on the processor architecture as well as on the used processor.

Secondly, every workload has a different instruction mix and different I/O characteristics. As QEMU delays interrupts and does not count the idle time, every workload progresses with slight different speed. To find a representative correlation between cycles and elapsed time, we measured the runtime of our simulation on the hosting system, set it in relation with the real runtime of the workload, and calculated the average instruction execution time, which ranges for computational intensive workloads between 1.9 to 4.3 nanoseconds. For the remainder of this work, we assume an execution time of 3 nanoseconds per instruction. The pace of our simulation is therefore about 24.3 MIPS.

6.3.2 Temporal Sampling

To overcome the vast amount of memory required to hold the history of every page frame, we sample our traced data with a granularity of 40,000,000 simulation cycles, which equals about 120ms. We determined this value empirically, as it showed best trade-off between accuracy and relevance. For this purpose the memory reconstruction and system state recovery, as described in Chapter 5, reconstructs both states till it reaches the end of a sample interval. After a sharing opportunity analysis, all states are combined to form our *frame history representation*. The frame history of each workload is the basis for our further evaluations. If a different sampling rate is used, it is explicitly noted.

6.3.3 Zero Page Frames

Previous research often excludes zeroed pages from their analysis [36, 41]. However, excluding them reduces the overall sharing potential of many workloads. There are two reason to exclude them. Firstly, if the operating system or an application actively zeros pages to prepare them for future use, e.g., Windows [43], and

merging them would introduce undesirable overhead on the first write. Secondly, if an analysis works on pages, rather than on page frames, and cannot distinguish between a zeroed page frame and a dedicated *zero page*. A zero page is a special page frame found in Linux [7], which is mapped as a COW page into processes to provide zeroed memory. The actual memory is allocated only if a write access to such a page occurs.

In general, a memory scanner should not ignore zero page frames, as many applications allocate memory, modify it, and reset the content back to zero. These intentionally zeroed pages do not point to the zero page frame anymore, instead redundant zero page frames have been created. To reduce the memory footprint of VMs it is most advisable to merge these pages back to the zero page.

To get an idea of the quantity of zero pages in various workloads, we performed several benchmarks. Our results suggest that most zero (guest) page frames are intentionally zeroed and many remain in this state for long periods of time. Table 6.7 shows the average amount of zero page frames and the average intra-domain sharing potential in five workloads. The zero page frames are even more important, if no matching candidates for other pages can be found or inter-domain sharing is prohibited due to security restrictions. The temporal development of zero page frames during a kernel build can be seen in Figure 6.2. It also shows that zero pages occur in all allocation groups. Zero buddy and page cache pages are stable, and remain unchanged over time.

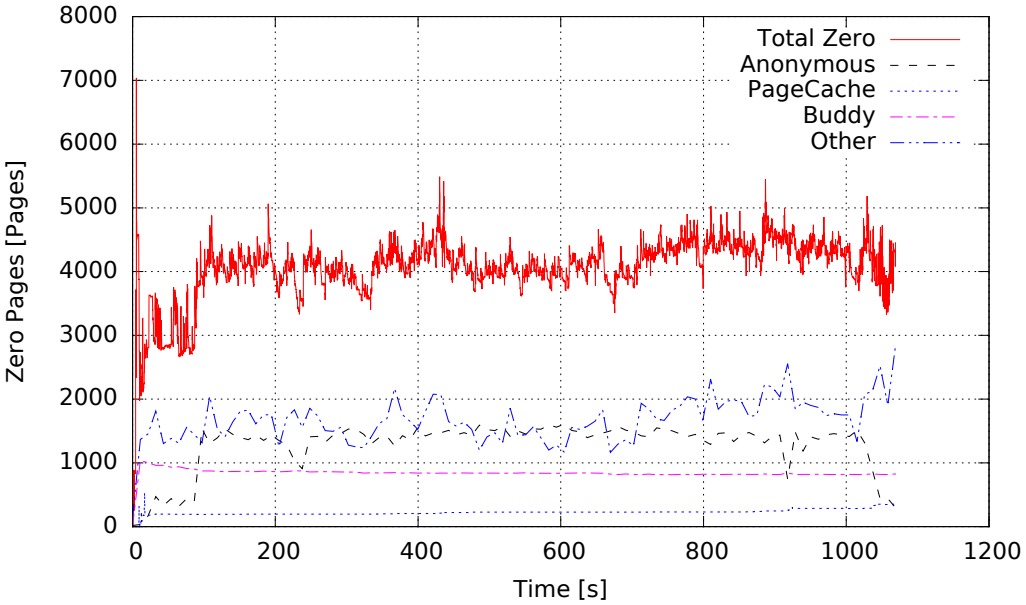

Figure 6.2: The temporal development of zero page frames during a kernel build enriched with semantic information. Page cache and buddy pages are stable for long periods of time.

Workload	Zero Pages	Intra-domain Sharings	Percentage
KB1024	4063	7410	54.83%
KB256	3231	6994	46.20%
B++	3067	98350	3.12%
PDB	3010	7277	41.36%
GCC	6880	22214	30.97%

Table 6.7: The average amount of zero page frames and the average intra-domain sharing potential of five exemplary workloads.

6.4 Sharing Opportunities

To classify the sharing potential of different workloads, we start with a sharing opportunity analysis, considering two different runs of each workload. That allows to correlate our findings with other studies and shows the sharing potential of our workloads.

6.4.1 Sharing Potential

As different workloads show different page duplication characteristics, we chose three workloads, which greatly differ in their amounts of mergeable pages and their progression over time. The temporal development of mergeable pages of *Bonnie++*, a *Linux kernel build* (KB1024), and *SPEC 403.gcc* are depicted in Figure 6.3. We chose these three benchmarks, to cover different aspects of sharing characteristics. The kernel build is an I/O intensive workload, which only alters few page cache pages through content modifications, it primary adds new content. In contrast, bonnie++ excessively creates, stores and reads data from disk. The 403.gcc benchmark in turn is memory-bound and does not change disk content.

Our freshly booted 64 bit Ubuntu server installation offers a sharing potential of about 15,000 pages (58 MiB) excluding the page cache. If the page cache is included the sharing potential increases up to approximately 50,000 pages (195 MiB). This nearly doubles the amount of sharing potential found in previous research [4, 41]. The difference can be explained by two facts: Firstly, previous research explicitly excluded zero page frames, which we do not, and during system start-up about 4,000 zeroed page frames (16 MiB) exist. Secondly, the greatest amount of additionally mergeable pages stem from our 64 bit system and its applications, which not only occupy more main memory than the 32 bit systems used in previous work, they also have bigger image files occupying more page cache frames.

The residual sharing potential stems from the executed workload's operations. Each workload has its own characteristics, e.g., the kernel build adds almost con-

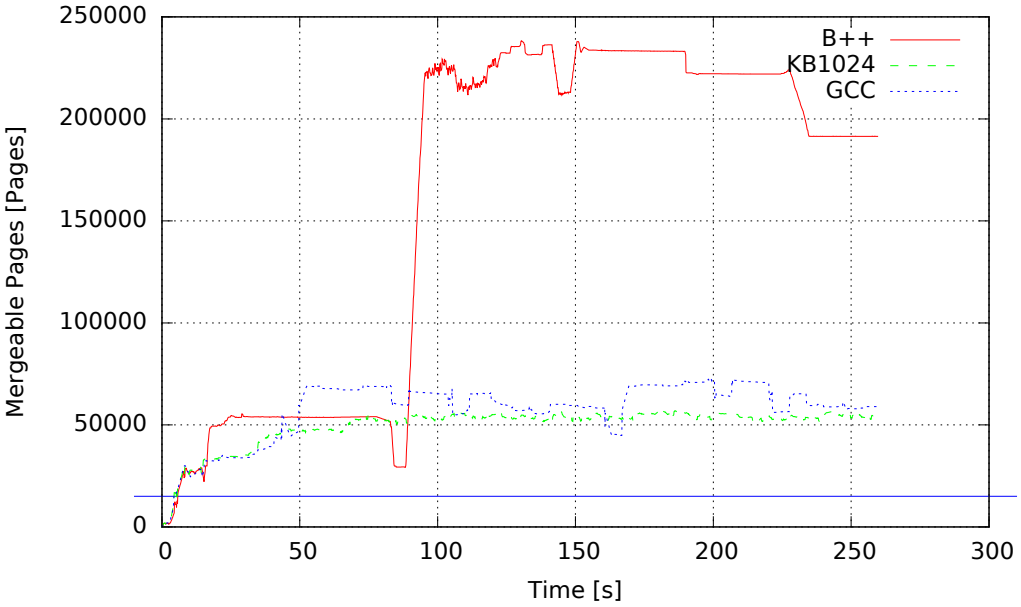

Figure 6.3: Overall sharing potential of B++, KB1024 and GCC over time. The line indicates the base sharing potential of our Ubuntu environment—about 15,000 pages (excluding the page cache).

stantly 25,000 pages (97 MiB) to the sharing potential. Whereas, only a retiring *bonnie++* (around 150 seconds) allows nearly all page frames previously used by bonnie++, i.e., about 200,000 (781 MiB), to be shared across *different VMs*. In the second execution phase (between 80 and 150 seconds) the theoretical sharing potential is as high (approximately 180,000 pages), but so are the fluctuations in stable content. The 403.gcc workload is analyzed in more detail in Section 6.6. It shows typical sharing potential of memory bound workloads, as about 63% of it originates from other sources than page cache content.

6.4.2 Semantic Information

The initial declension of sharing potential (around 80 seconds) in the bonnie++ workload indicates the beginning of the second execution phase, as all available main memory is allocated to bonnie++'s hosting task. Therefore, Linux has to release the buffered page cache content. Figure 6.4 shows the semantically enriched sharing potential of bonnie++. Enhanced with semantic information bonnie++'s stabilization and increase in sharing potential can be explained. The actual internal changes of the page cache (around 150 seconds) are not visible in the overall sharing potential and from outside a hardware-accelerated VM without introspection

support. However, the associated semantic information from the OS introspection allows to observe this behavior. The source of nearly all mergeable pages stem from already released page cache frames, rather than from actually used pages. For instance, on Windows this sharing potential will decrease fast after de-allocation, since Windows actively zeros freed memory [43].

Memory deduplication, which tracks memory allocations, might either fail to merge the sharing potential as the memory is freed, or detect this change and replace every released page frame with the zero page of the host. In the latter case, the memory footprint would be directly minimized, beyond the size which would be achieved with regular merging, since memory content is typically less redundant.

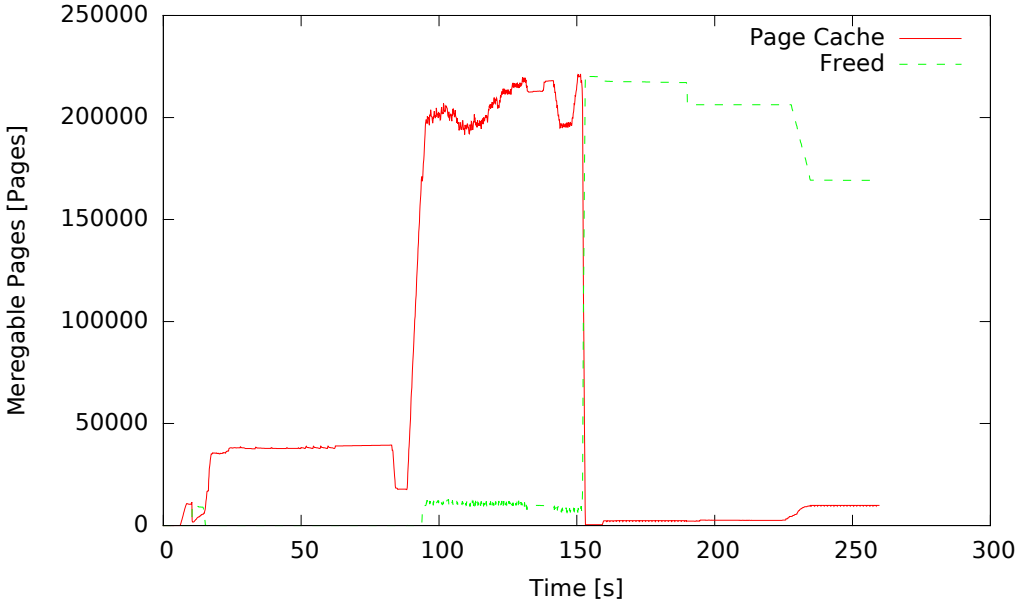

Figure 6.4: The semantic enriched sharing potential of bonnie++. After it has finished the second part of the benchmark, it deletes the created file content (around 150 seconds).

Stability A prerequisite to harvest duplicate memory pages is *stability*. Furthermore, stability is the only statistical property of page frames that we found to tell sharing opportunities apart from regular unsharable pages. A page is considered stable if its content has not changed within a chosen time interval. Merging unstable pages, even if they have a sharing partner, computation resources are dissipated.

Our benchmarks show, that finding a sharing candidate depends on two different criteria. Firstly, a workload must produce identical content. For example, during a Linux kernel build, the probability of finding identical content is high, because two workloads (in different VMs) load identical file content from background storage and produce identical output files.

Secondly, the produced or loaded content has to remain stable for long periods of time, otherwise, even if a memory deduplication system could instantly merge identical pages, their content might change immediately and the sharing is broken.

Thus, to share pages across domain boundaries or between instances of applications, the temporal divergence must be small for short living or unstable pages. For example, if a page is stable for about 10 seconds, but the other workload runs with a temporal differential of about 9 seconds, the sharing period is reduced to 1 second, and typically does not justify merging.

6.5 Memory Access Analysis

Our recorded benchmark traces allow to observe the spatial and temporal development of page frames. Memory access frequencies and access patterns can be inferred during memory reconstruction and stored in sampled history files.

6.5.1 Memory Write Frequency

To focus a memory scanner on rarely changing or stable page frames, a frequency of memory accesses can identify potential candidates. As a page frame's content is not changed by load operations, the frequency can be limited to write operations. We calculate the memory write access frequency on sampled time intervals of about 120 ms (40,000,000 cycles). To calculate the frequency for a point in time, we use a window of 13 intervals (it covers about 1.5 seconds of simulation time). Figure 6.5 depicts the frequencies before a page frame becomes stable for at least 1.5 seconds for B++, KB256, and PDB. All evaluated workloads show similar write frequencies, even though they perform completely different tasks.

The database workload (PDB) indicates stability slightly better than others. About 89.86% of all stable frames in this benchmark are preceded by a low write access frequency (one to four accesses within our sliding window). On average about 77.95% of page frames in all evaluated benchmarks show this behavior.

Memory write frequencies allow a memory scanner to include pages with low frequencies into its scan process and temporarily exclude pages with high frequencies. However, a memory scanner should not exclude pages *permanently* solely on the access frequency observed at some point in time. Instead, it should still

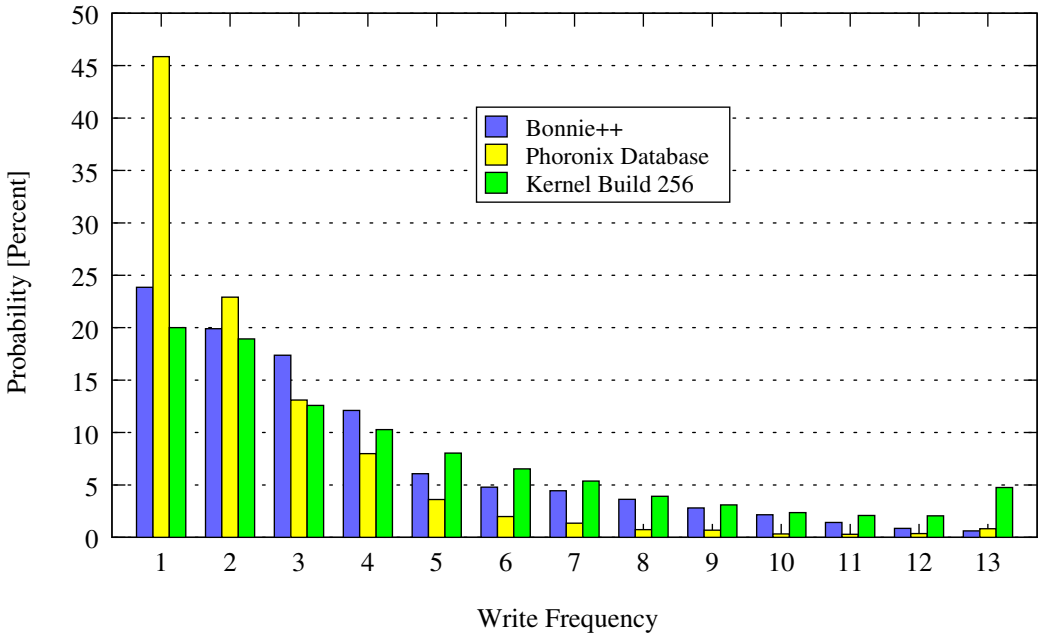

Figure 6.5: Memory write frequencies for three different workloads with the probability to be observed before a page frame becomes stable. All evaluated workloads show similar distribution.

monitor the access frequency of excluded pages and adapt scan areas appropriately. Especially, as soon as the semantic background of a page changes and it is used for another purpose, there is a high chance that its access frequency changes also. The page could then become a viable sharing candidate again.

In disk power management one strategy to determine a good point to bring the device into a low power state can be determined by a correlation between busy and idle times. It has been found that short busy phases are typically followed by long idle phases. The same correlation between busy and idle times can be done with the write access frequency on memory pages to provide a memory scanner with a policy of when to include or exclude pages. We consider a page frame busy, if it is written within every time interval (120 ms) with a grace period of 2 intervals. Otherwise, if it remains unchanged, it is considered *idle*. Figure 6.6 shows the relation between busy periods and the following idle periods of a Linux kernel build equipped with 256 MiB of memory. The diagram shows that long busy phases are never followed by long idle phases and thus page frames cannot become stable. Such page frames showing this access characteristic are most likely used for frequent device DMA or contain stacks and dispatcher structures of the kernel, which are constantly modified. However, it is worth to include pages

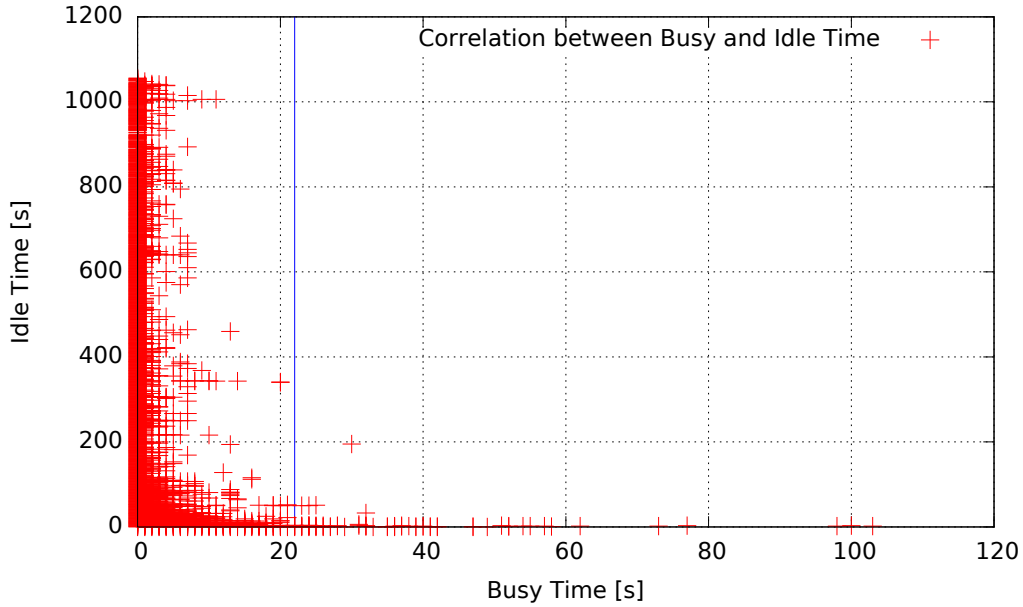

Figure 6.6: Page frame busy times and their following idle periods. Page frames with high busy times (more than 10 seconds) never become idle for long periods of time.

with a short busy phase. Nearly all page frames (about 45,000 pages, 175 MiB), which idle longer than 10 seconds are associated with the *page cache* of Linux. About 39,000 page cache pages (152.34 MiB) can be shared with another VM executing the same benchmark.

All evaluated benchmarks show similar relations between busy and idle time, they only differ in the length of busy and idle periods. We observed that idle pages associated with the page cache experience only short busy phases at the beginning of their lifetime. Therefore, a memory scanner should focus on these pages and check pages with high access frequencies and long busy times less frequent or exclude them with a simple threshold. In our benchmarks a page that was busy for more than 15 seconds never became idle before it is reallocated or freed.

6.5.2 Memory Access Patterns

Our analysis of more complex temporal access patterns, e.g., the temporal sequence of writes, did not find correlations between stability and sharability, not already covered by the memory write frequency. For instance, a typical pattern of stable (page cache) pages, is one (DMA) write access—i.e., an access frequency of 1—followed by a long stable phase. All other patterns showed no new infor-

mation to predict the future development of page frames, neither helped to identify stable pages better than with the memory write frequency. On the contrary, probabilities and patterns degenerate to noise and make predictions like equally distributed random numbers. We conclude that it takes either a more complex analysis scheme or that temporal access patterns are unsuitable at all. However, a more complex analysis scheme, such as neuronal networks, might be too time consuming to be used for memory scans.

6.5.3 Conclusion

We discovered that more complex patterns such as the exact write sequence differ with every workload and even within it. They further showed no clearly observable correlation with stability or sharability in our benchmarks. Patterns not already covered by access frequencies, degenerate to noise.

Thus, we conclude that the *memory write frequency* with a reasonable threshold, i.e., about 4 accesses within a window of 1.5 seconds, can help classifying page stability and indicate a stable page frame. Pages with high memory access frequencies can be excluded permanently from a scan, if their overall busy time exceeds a certain threshold. For our evaluated benchmarks it is about 15 seconds. To further improve such scheme, internal semantic information can help to re-include an unstable page if its allocation has changed. Memory access frequencies and busy times can be observed in real systems, through an inspection of dirty bits set by the *memory management unit* (MMU) to indicate a write access to a page.

6.6 Semantics and Stability

We discovered that the stability of pages can be inferred by monitoring memory access frequencies. However, the lifetime and development of stable pages requires more sophisticated methods. In this section, we want to evaluate if there also exists correlations between a page's stability and its semantic background. Finding such correlations could help focusing memory scanners by permanently excluding whole ranges of pages with certain semantics and limiting monitoring of write access frequencies to more promising page types. Our evaluation is based on our proposed HMM. We chose five exemplary workloads which cover a wide-range of execution characteristics.

Figure 6.7 shows probability of a page to be allocated for a certain purpose found in each workload. Freed or reallocated page frames are omitted, since those

pages are either rarely found (less than 1%) during workload execution or are instantly reused.

Every workload has a unique memory allocation profile, which varies within identical workloads only equipped with different main memory size (KB1024 and KB256). It shows on the one hand, that our workloads cover a wide-range of different memory allocation characteristics, but on the other a single heuristic based on the data of multiple workloads most probably will not provide good predictions. A heuristic can only be estimated for each workload and for a specific configuration. On average these workloads allocate about 25% page frames for the page cache (already covered by I/O-based hinting) and 35% frames for anonymous memory, which must be harvested by linear memory scanning. A specific memory allocation heuristic can be used to identify workloads and adjust the scan behavior of a memory scanner.

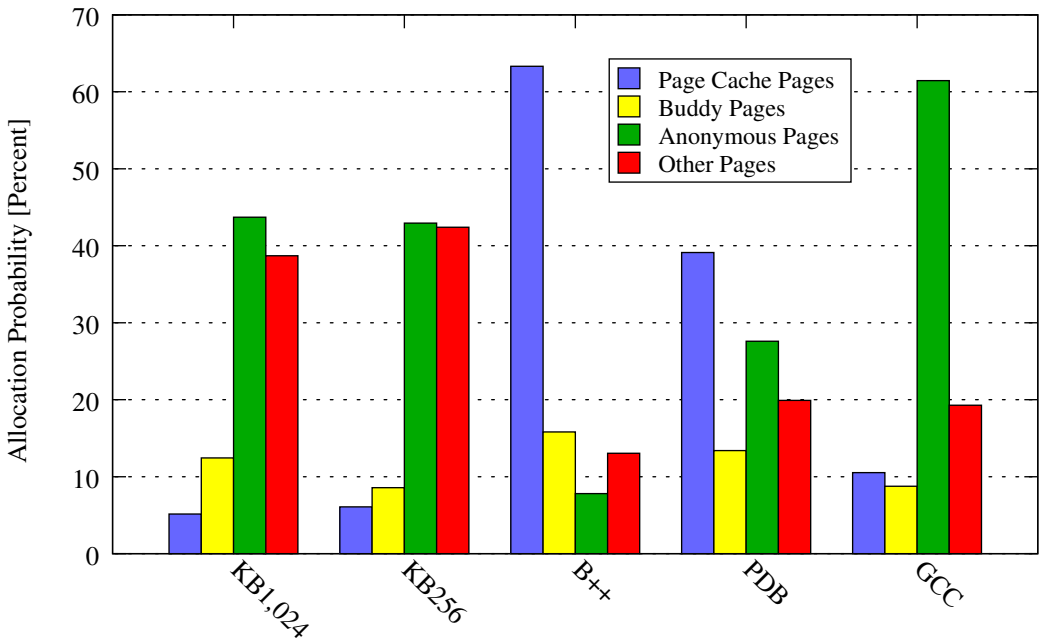

Figure 6.7: The memory allocation probabilities for five exemplary workloads. It shows that a general heuristic to predict allocation behavior is unrealistic, if a mixture of different workloads are executed.

Stability Our HMM predicts the future development of page frames. Figure 6.8 shows the (transition) probabilities for page frames to remain in stable states for each memory category and examined workload. In all workloads the probability for a page to remain stable once it has been loaded into the page cache is about

85% to 98%. That explains why I/O hinting achieves good sharing results. In contrast, the stability of modified or created page cache content is less significant. In a kernel build about 50%, without memory pressure (KB1024), and 30%, with memory pressure (KB256), of pages remain stable for more than 3 seconds. Our database workload (PDB) reduces the probability even to 5%. The results therefore suggest that for certain workloads, page cache pages holding modified or new file content should not be targeted by memory scanners, because these pages change frequently. The average stability probability of *anonymous memory*, one

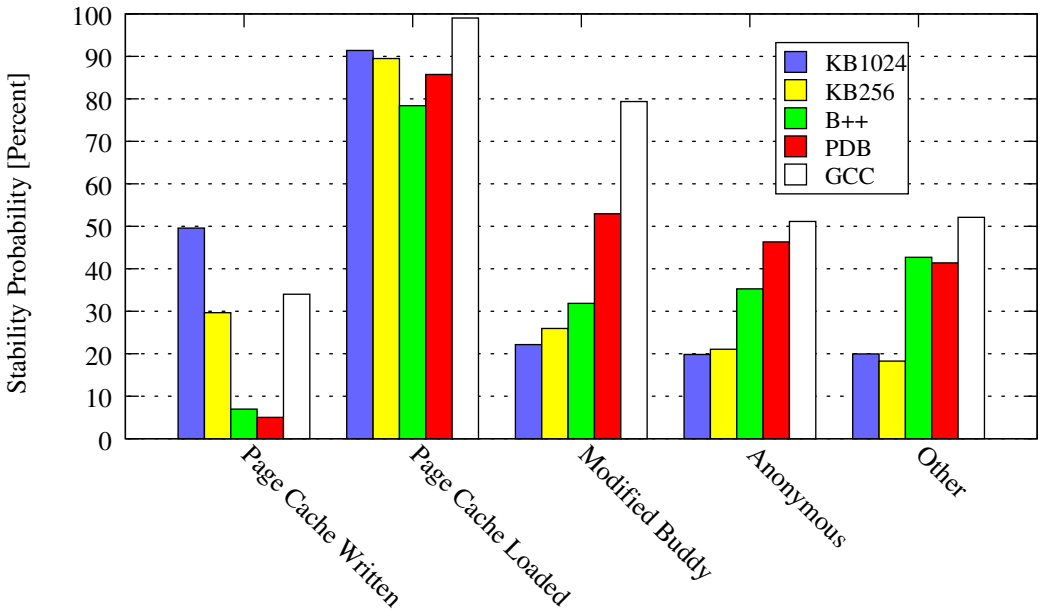

Figure 6.8: Stable page frames states and their probabilities to remain in this state. A page frame considered stable has remained unchanged for about 3 seconds.

possible source of additional sharing opportunities, is on average about 35%. For kernel builds only about 20% of anonymous pages remain stable. The low stability probabilities are explained by the compilation process itself. Every gcc instance compiles a source code file and terminates after it has emitted the corresponding object file. Thus, all anonymous pages are released and instantly re-allocated to other instances. OS introspection data shows that most of the *stable* anonymous memory pages (about 90%) do not take part in building the Linux kernel, instead they are allocated by idle system applications.

As the variance of nearly all categories depends on the workload, predicting a general suitability of semantic page groups is only possible for every workload

individually. Predicting different workloads with the same HMM lead to equally distributed transition probabilities and reduces the predictability of future page frame usage to pure guessing.

The most stable page content stems from loaded files, which seconds the findings of many research projects [4, 36, 41]. It further suggests that concentrating on load operations, might improve hinting based on I/O operations, at least, for some workloads, e.g., databases, even further. Thus, an initial workload type estimation, adaptions to the scanners behavior, or manual configuration might help, to harvest more sharing opportunities faster.

Stability and Sharability Stability probabilities predict the future development of a stable page without considering its history. To further investigate the amount of stable pages and correlate them with sharing potentials, we recorded state changes of our HMM over time. This recorded history forms a similar representation as our frame history (see Section 5.3), but includes state information, rather than write counts.

In the following we will concentrate our investigations on the SPEC 403.gcc benchmark. It is a memory bound benchmark with an *anonymous memory* stability probability of 51%, and thus represents typical sharing and stability potentials.

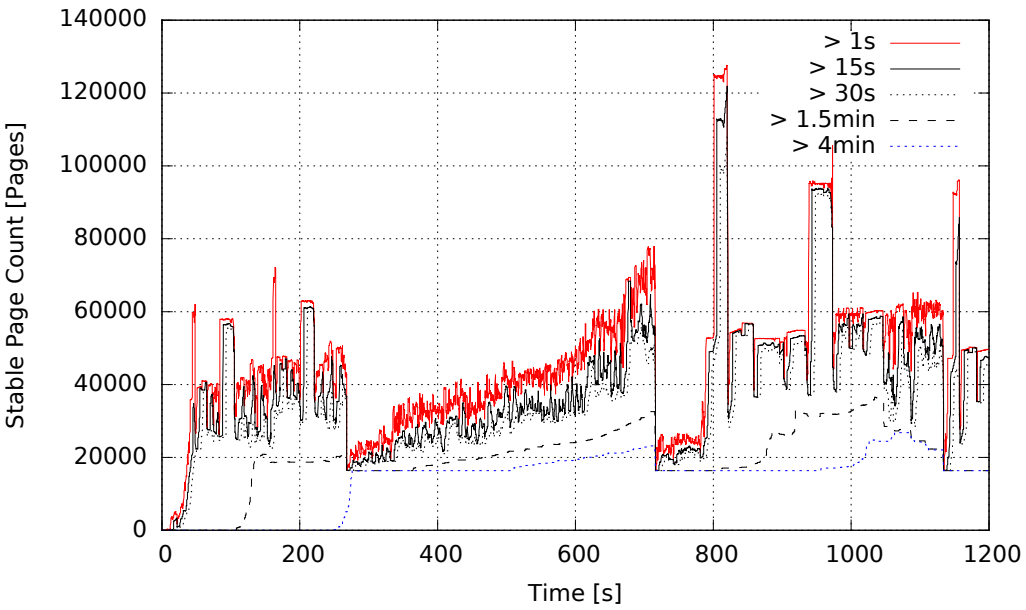

Figure 6.9: The temporal development of stable anonymous memory pages in the SPEC.403.gcc benchmark. Depicted with different stability criteria.

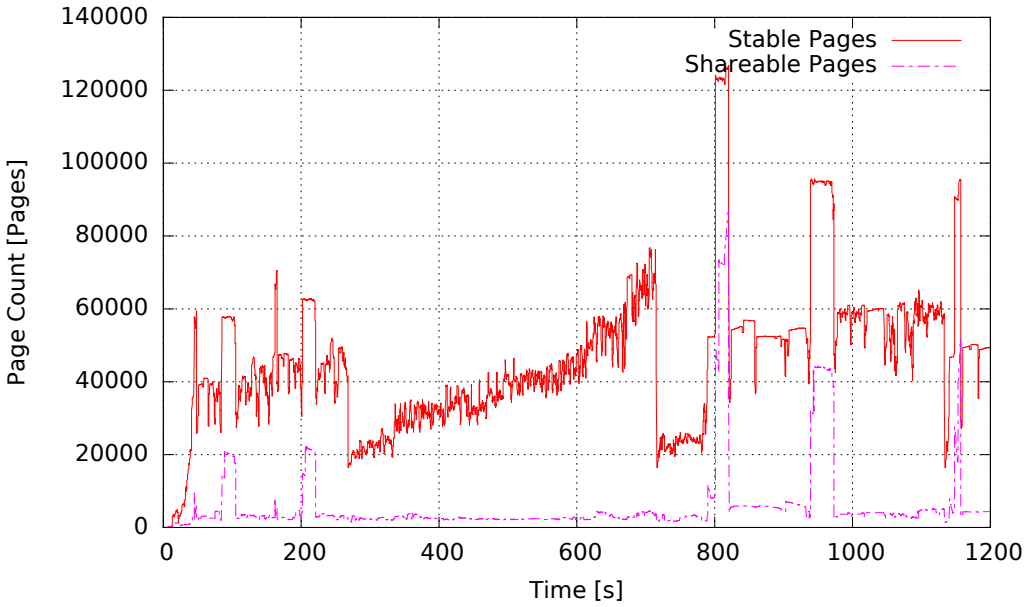

Figure 6.10: Although a large amount of pages are stable, they still cannot find a sharing partner, not even within another VM, executing the same workload with virtually no temporal displacement.

Furthermore, only 21% of the theoretical sharing potential of 403.gcc originates from loaded file content, the remaining sharing opportunities stem from kernel and anonymous memory and must be detected through scanning. Figure 6.9 shows its temporal development and the amount of stable pages based on different stability criteria (from 1 second to 4 minutes). The diagram shows the different execution and memory allocations phases of the simulated compilation process.

In the execution phase between 280 and 720 seconds the amount of stable pages slowly increases from 20,000 to 40,000; on average 30,000 pages (117 MiB) excluding the fluctuations. Unfortunately, during this slow changing period only about 2,400 pages (9.37 MiB) can actually be shared, even with an identical workload running in parallel with nearly no temporal difference. The temporal development of sharing opportunities and stability can be found in Figure 6.10. Only after a radical change in the workload (805 seconds) about 42,000 pages (164 MiB) can be used for *intra-domain sharing* as in this phase previously created data is replaced with identical patterns.

Considering the whole workload, such states only last for short periods (less than 20 seconds) and merging should be avoided to conserve computational power. That further suggests, if anonymous memory becomes stable at last, it shows less sharing potential than pages originating from disk. Therefore, a scanner aware of

OS internal allocations, should be configured to exclude guest anonymous pages, for CPU- or memory bound workloads, as the produced data is fast changing and small temporal differentials make (inter-domain) sharing unfeasible. However, if the CPU utilization for memory bound workloads decreases, the probability to identify stable pages and share memory content rises.

To further investigate stability, we applied various stability criteria to 403.gcc. The resulting temporal development and amount of stable pages can be found in Figure 6.9. As memory deduplication is always a trade-off between invested resources and resulting gain, we chose different time periods, which consider a page frame as stable. They range from 1 second, a unrealistic fast and expensive scan, to 4 minutes which is a realistic scan period of KSM [35]. The different stability criteria show the limitations of slow linear memory scanning, and further how important it is to choose a scan ratio carefully. The decline in the amount of stable pages discovered is proportional to the chosen time interval, the longer it takes to consider a page stable, the less pages are discovered: The decline can be approximated by a *linear function* for all examined workloads – except for the chaotic phases of SPEC.433.milc (see Figure 1 in the appendix). A doubled time interval nearly misses half stable pages, in the best case a scanner would also miss half of

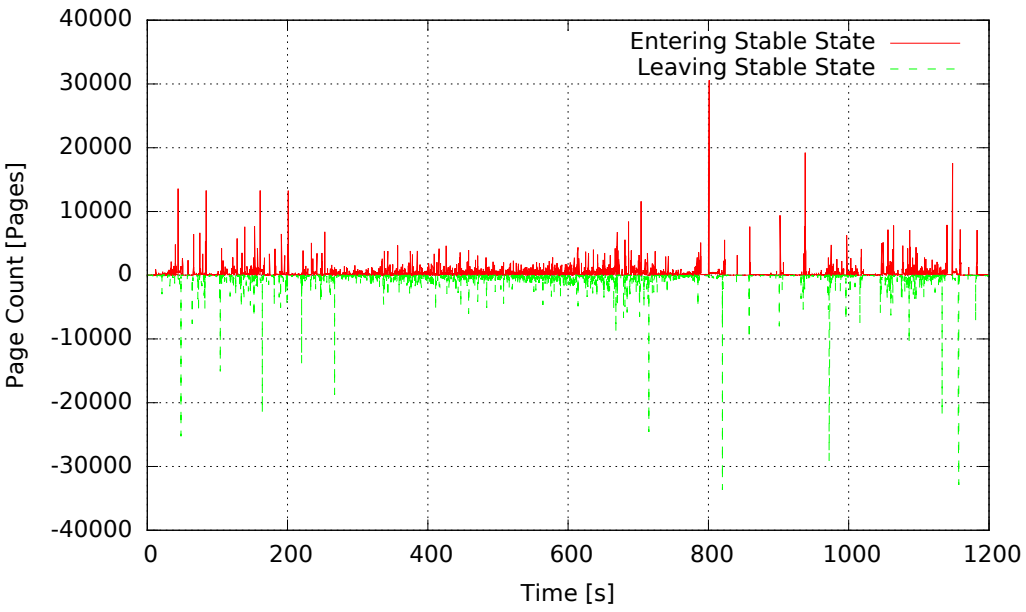

Figure 6.11: The amount of pages entering or leaving a stable anonymous memory state in the HMM, depicted over time. The peaks indicate a common, but unstable behavior of anonymous memory.

the sharing opportunities. Furthermore, the amount of changing pages in SPEC 403.gcc is not constant and contains bursts, which makes scanning even more ineffective. Figure 6.11 illustrates the amount of pages becoming stable and entering and leaving this state over time. Even though the absolute amount of pages is increasing (in the interval between 280 and 720 seconds), many pages (about 10,000, 39 MiB) leave and enter this state every second. These large fluctuations show that memory deduplication is nearly impossible for this workload. The same is true for all evaluated workloads actively working on anonymous memory. Only idle applications, e.g., desktop workloads, or applications with small working sets (e.g., SPEC 464.h264ref) might benefit from merging anonymous memory.

A sensible stability criterion seems around 15 seconds for anonymous memory pages in this benchmark as it shows the best trade-off between amount of scanned pages and sharing opportunities. Unfortunately, giving an ideal scan ratio for all workloads is hardly possible, as already deducted by previous research [11]. The optimal scanner configuration depends to a great extent on the characteristics of the executed workloads.

Stability and Semantics We want to investigate the correlations of semantic information and stability, as semantics, e.g., page cache hints, has already been proven to be valuable information.

We can confirm that page cache pages remain untouched and outperform every other page category in stability as well as sharability, if not used for memory mapped file management (as in our database benchmark). Their content has a high probability (about 80% in a kernel build and 98% for 403.gcc) to be identical, since they are loaded from similar virtual disk images and remain unmodified.

That is true for all workloads considering only loaded content. However, that is not true for other page categories, such as buddy and anonymous pages. Figure 6.12 shows the smoothed temporal development of stability and sharability. Although the amount of stable buddy pages is nearly constant and high, about 8,000 pages (31.25 MiB) during a kernel build, only about 20% (6.25 MiB) of these pages can be merged. These page frames are occupied by the Linux kernel image, as loaded during system boot. Buddy pages show this low sharing rate in all evaluated workloads, although they are stable. The remaining buddy pages cannot find a matching candidate, not even in other VMs, running the same Linux kernel. We believe that this is caused by the more realistic and non-deterministic execution of QEMU's simulation.

The shareable amount of anonymous memory is higher with about 50% in the peak phases, but it rapidly drops to 11%. The change in stability and sharability over time makes merging anonymous pages difficult and explains why many deduplication mechanisms fail to harvest all measured sharing opportunities, since

they only exist for short periods of time, but contribute to the overall amount of shareable content. Although, a memory scanner can focus on these stable pages, it still has to scan them all before a sharing can be established. Therefore, a successful merge also depends on the speed of the discovering mechanism, e.g., hashing and identifying a sharing partner, and might miss these relatively short peaks. Furthermore, it is not clear if such short living pages should be merged at all, as the overhead might easily exceed the benefits.

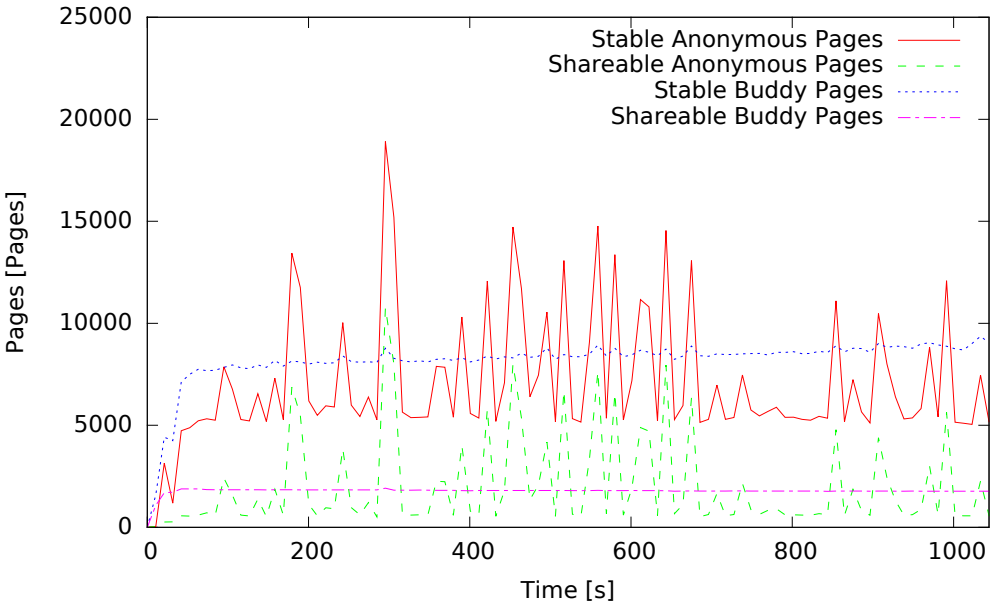

Figure 6.12: The smoothed temporal development of stability and sharability for buddy and anonymous page frames during a kernel build. Although the stability of buddy pages is high, only about 20% can be actually shared.

6.7 Write Working Set

The working set of a process consists of recently accessed and written pages [14]. However, memory content changes only if a write operation occurs, thus we propose the *write working set*. This working set contains all recently changed pages and should be excluded from scans. If a memory scanner keeps track of the write working set, it can focus its scan on all pages leaving it. That drastically reduces the amount of hashed or compared memory content.

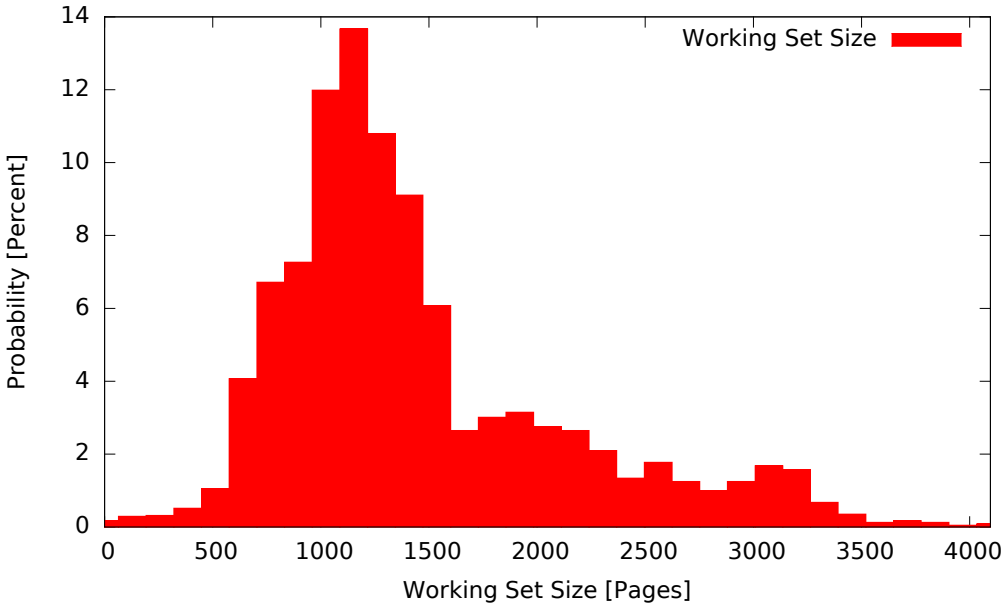

Figure 6.13: The write working set of VM running a Linux kernel build equipped with 1024 MiB of RAM.

KB256 shows small write working set sizes, it consists on average of 1,520 pages (5.93 MiB) in our considered time interval (120 ms). A typical distribution of write working set sizes can be seen in Figure 6.13. In KB256 on average 723 pages (2.82 MiB) leave the write working set, and only these pages must be further examined to find possible sharing opportunities. These pages have recently changed, but as the have left the working set they have *stabilized*. Estimating the write working set every 160,000,000 cycles (about 480 ms) yields on average 1,850 page frames (7.25 MiB) leaving it. Estimating the write working set every 160,000,000 cycles (about 480 ms) yields on average 1,850 page frames (7.25 MiB) leaving it.

Other benchmarks show similar behavior, except the SPEC.464.h264 benchmark, which has a tiny working set of about 758 page frames (2.96 MiB), where on average only 87 pages (0.33 MiB) leave. And SPEC.433.milc, executing a quantum chromodynamic simulation, tend to produce large working sets with typical peaks containing 85,000 pages (332 MiB) (caused by the gauge field calculation) and average fluctuation of 3,743 pages (14.61 MiB)[1].

As the stability of a page might still be affected by their internal usage, further information sources can be added to reduce the actually scanned pages, e.g., the

[1]Figure 2, presenting the working set distribution of SPEC.433.milc, can be found in the appendix.

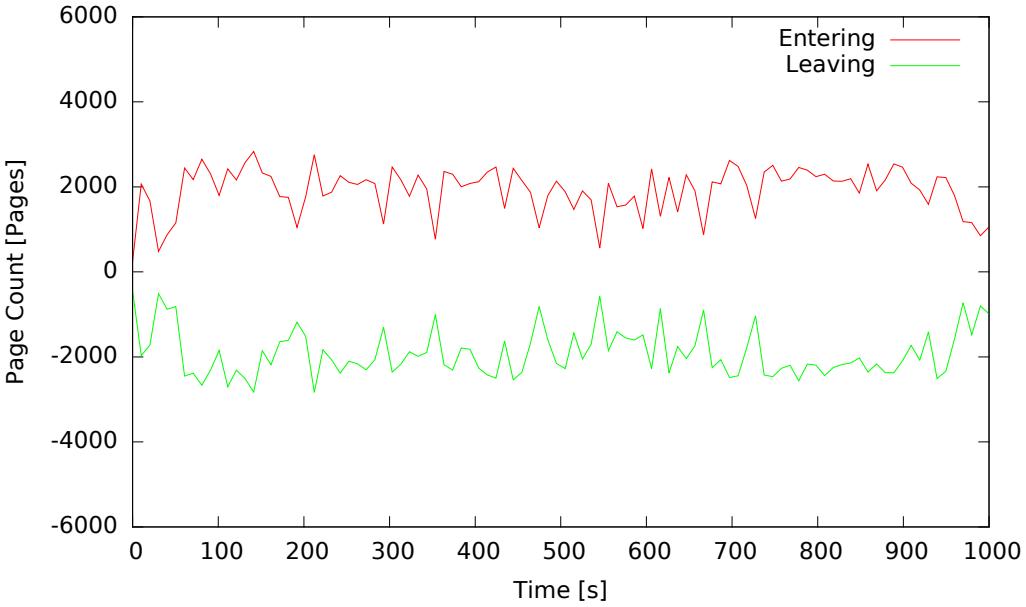

Figure 6.14: The write working set changes of a VM running a Linux kernel build equipped with 256 MiB of RAM.

recent write frequency or a stability predictor based on busy times. The working set changes depicted in Figure 6.14 are typical for all simulated workloads, the only exception is the working set changes of SPEC.433.milc (see Figure 1 in the appendix). Although on average about 1,850 page frames must be scanned to discover changed content, the amount of pages leaving a working set is not constant and can hardly be modeled statically. However, considering only pages leaving the write working set allows a memory scanner to focus on *stable* content and to reduce the actually hashed or indexed pages drastically.

6.8 Conclusion

We evaluated a wide range of different benchmarks, covering I/O- and memory bound workloads. Although each workload has unique characteristics, all have two properties in common. Firstly, loaded content from backing store remains stable with a probability of at least 85% and can in more than 80% be shared with other VMs loading identical content. Secondly, about 78% of all pages show a low memory write frequency (4 write accesses within a window covering 1.5 seconds) before they become and stay stable for at least the same time.

We conclude that using memory write frequencies on pages not associated with file caches could reduce the scan overhead and focus a memory scanner on

stable content. High write frequencies indicate unstable content and such pages should be *temporarily* excluded, till the frequency decreases. If a page is *busy* (constantly written) for more than 15 seconds, it can be *permanently* excluded from scanning.

Our evaluation of memory access patterns showed no meaningful correlation between a distinct pattern and stability or sharability. Furthermore, our HMM can only predict the stability of pages, if a specific workload type is known in advance. Thus, predicting the future usage of a page frame can only be done with memory access frequencies or busy time considerations.

We concentrated our research on sharing opportunities not yet merged by memory scanners. We considered their semantics and the temporal development of stable pages. Our evaluation revealed that the sharing potential of stable anonymous memory varies between 11% and 50%. However, as long as the workload is performing operations, shareable pages changing far too fast to justify merging.

Additionally, we evaluated write working set changes to focus a scanner only on recently changed content, rather than examining all pages belonging to a VM or application. A write working set can show fluctuations, but on average the amount of pages leaving a working set stayed within reasonable limits (723 pages in 120 ms, 2.82 MiB). Only these pages must be scanned to keep track of stable memory content, which are 50% of all recently changed pages (1,520 pages, 5.93 MiB).

However, it is not clear how much overhead the working set or the write frequency estimation generate on a real system. To estimate these characteristics, a page table walk, collecting the page flags to indicate changed page content, is necessary. However, as memory scanners otherwise consider all pages allocated to a VM or application, it might justify the effort.

Chapter 7

Conclusion

In this work, we analyzed sharing potential, which is not yet sufficiently harvested by memory scanners, with the goal to rank pages based on the likeliness to yield sharing opportunities and include or exclude them from scans. We investigated the correlation of memory access frequencies and patterns, page stability and further semantic information with memory duplication.

To acquire all write operations and OS introspection data for offline analyses, we improved Rittinghaus' [41] original framework for trace acquisition and processing by strictly separating data recording from analyses. Our design supports flexible offline analyses of sharing opportunities, memory access frequencies and access patterns. Our changes also increase scalability. We have extended the original framework further, to acquire semantic data by introspecting recent 64 bit Linux guests. We deduce memory content, from the recorded memory modifications. Recording long-running workloads such as SPEC benchmarks is feasible in our system. In a secondary step, we sample the memory image, infer sharing opportunities, and fuse this information to a compact history of each page frame enriched with semantic information. With the fast functional simulation and extended toolset, we were able to retrieve different trace files and analyze various workloads.

We simulated workloads that cover different program characteristics from I/O dominated real-world benchmarks to synthetic filesystem stress tests and database test suites. Our analyses show that sharing potential heavily depends on the executed workload. Predicting the future of pages or directly inferring shareable content seems impossible, as different workload configurations, e.g., kernel builds with variable memory sizes, show varying stability and sharability characteristics.

We discovered two general optimizations for linear memory scanners to avoid futile scans. Firstly, a memory scanner should only regard pages that show a low write access frequency, since that indicates stable page content in the near future with a probability of 80%. In addition, busy times or memory semantics can

be used to exclude page frames permanently from scans, e.g., device associated memory or kernel stacks. Secondly, a scanner should focus on pages leaving a write working set instead of linear scanning all pages. These two optimizations can be used together, since both can be implemented with a page table walk and minimal extensions to the scan infrastructure.

7.1 Future Work

Although our data acquisition framework and QEMU performed well, and allowed to collect all required information, future versions might still be improved with more efficient compression mechanisms, faster memory hooks, and dynamic OS introspection.

Trace Infrastructure Improvements A limitation of the current storage server is the compression ratio, which varies from 1:4 up to 1:400, depending heavily on the consecutive write accesses, their addresses and data. A future version might contain an improved compression scheme, solely designed for patterns found in memory accesses. The main problem of zLib and other common compression techniques is their general usability, ignoring specific data patterns. A compression scheme, as suggested by [10], utilizes such patterns and provides dedicated predictors for addresses and for the accessed data.

Another limitation of our storage server is that it currently runs on a single computer node only and cannot utilize the computation power of server clusters, which is necessary if parallel simulations were used.

Simulation Our implementation of *memory hooks* performs a guest page table walk to retrieve physical addresses of a memory operation. This reduces the maximum achievable simulation speed. A future implementation could directly use the address translation layer of QEMU, which would make an additional address translation unnecessary, as the virtual TLB maintained by the softMMU already contains the virtual to physical address mapping.

QEMU is missing different features, e.g., a parallel simulation and code generation. That might further speed-up tracing, if more cores can translate and execute code in parallel. Furthermore, if simulations should only execute few architectures, for example, x86 and ARM, it might be beneficial to implement new features into the QEMU's intermediate representation and allow a direct utilization of hardware features such as MMX, SSE, or Neon.

Recent versions of QEMU in conjunction with the tiny code generator (TCG), which is used for full-system simulation, suffer from multiple problems. Firstly,

QEMU is not able to start an Xserver from an Ubuntu distribution, even with different configuration the Xserver task always crashes during start up. Secondly, when QEMU uses the `icounter`, several Linux distributions livelock, progress is only possible after an interrupt created by an external device, such as a keyboard. That makes unsupervised booting and tracing impossible. In consequence, future versions of QEMU can only be used for simulation if these problems are fixed.

A different simulator could be integrated into our tracing infrastructure. That is possible since only a small intermediate layer must be implemented to comply with our storage library interface.

OS Introspection To become independent of a single modified OS kernel, a more general solution would be helpful. Different OS introspection mechanisms have been proposed for VM migration, which dynamically analyze the running OS and extract the required information with symbol files [12]. Another approach might use breakpoints during simulation to trace operating system events. Such mechanisms might augment a Linux and a Windows kernel on-the-fly. This would allow to use unmodified kernels even without inserted drivers to collect the OS system state, with its process creation events, allocations, and page table modifications.

Memory Scanner Improvements Our evaluation shows that harvesting duplicate memory pages is feasible for memory content loaded from disk.

However, anonymous memory is hardly shareable, as most applications produce data patterns, which if shareable at all—lasts only for less than 3 seconds. However, if a memory scanner initially inspects the address space of a freshly booted VM, it can permanently exclude page table memory, kernel stacks, and device associated pages from scans. Afterwards, it should only consider pages leaving the write working set. That focuses the scanner on actually changed page content and excludes all other pages from a thorough examination, e.g., hashing a page's content. Ideally this mechanism is combined with a memory write frequency. If a page's frequency is low the chances are high (about 80%) that the page will become idle for a period worth of merging. Combining working set and frequencies seems an ideal combination, since both attributes can be inferred by inspecting dirty bits. These can be obtained either by directly walking the hypervisor's page table or by inspecting the corresponding guest page tables. An implementation of such mechanisms should be evaluated to classify the usability of our proposed improvements.

Appendix

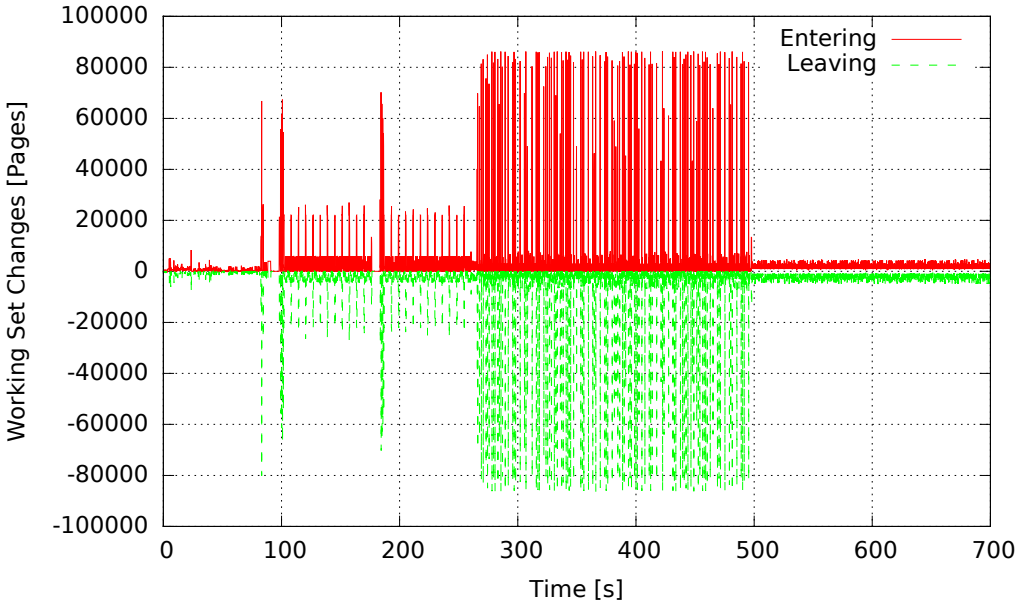

Figure 1: The write working set changes of a VM running SPEC.433.milc (1024 MiB memory).

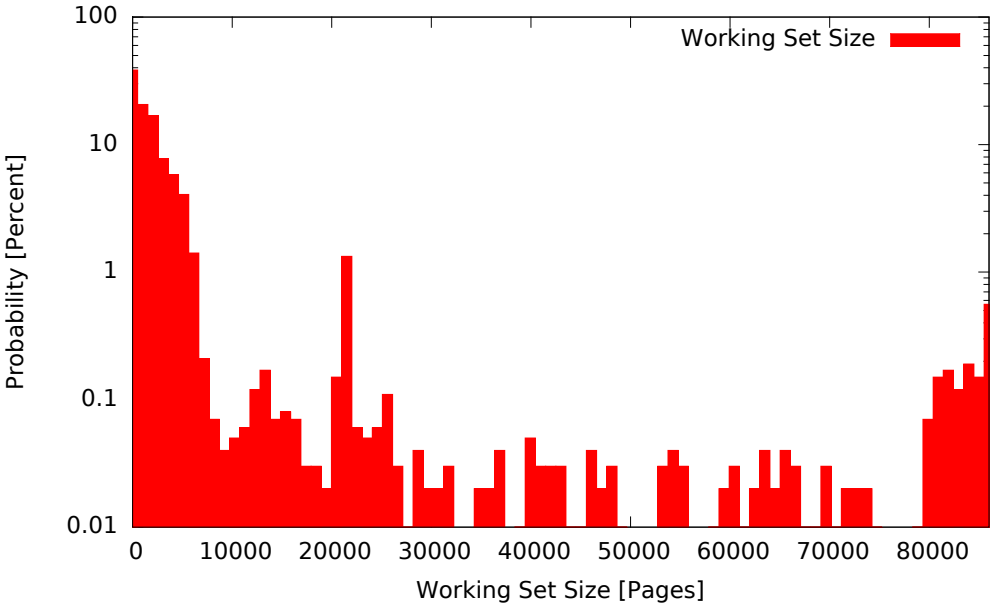

Figure 2: The distribution of write working set sizes of SPEC.433.milc.

Bibliography

[1] Keith Adams and Ole Agesen. A comparison of software and hardware techniques for x86 virtualization. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ACM ASPLOS, pages 2–13, New York, NY, USA, October 2006. ACM.

[2] David Anderson. Red hat crash utility. `http://people.redhat.com/anderson/crash_whitepaper/`. Accessed: 2011-08-05.

[3] Andrea Arcangeli, Izik Eidus, and Chris Wright. Increasing memory density by using KSM. In *Proceedings of the Linux Symposium*, Linux Symposium, pages 19–28, Ottawa, Canada, July 2009. Linux Symposium Incorporation.

[4] Sean Barker, Timothy Wood, Prashant Shenoy, and Ramesh Sitaraman. An empirical study of memory sharing in virtual machines. In *Proceedings of the 2012 USENIX Annual Technical Conference*, USENIX ATC, pages 273–284, Berkeley, CA, USA, June 2012. USENIX Association.

[5] Fabrice Bellard. QEMU, a fast and portable dynamic translator. In *Proceedings of the 2005 USENIX Annual Technical Conference*, USENIX ATC, pages 41–46, Berkeley, CA, USA, April 2005. USENIX Association.

[6] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. The gem5 simulator. *ACM SIGARCH Computer Architecture News*, 39(2):1–7, May 2011.

[7] Marco Bovet, Daniel P. ; Cesati. *Understanding the Linux Kernel – from I/0 ports to process management; covers version 2.6*. OReilly, Sebastopol, CA, USA, 3rd edition, 2005.

[8] Patrick Brady. Anatomy & physiology of an android. In *Google I/O Developer Conference 2008*, Google I/O Developer Conference, May 2008.

[9] Edouard Bugnion, Scott Devine, and Mendel Rosenblum. Disco: Running commodity operating systems on scalable multiprocessors. *ACM SIGOPS Operating Systems Review*, 31(5):143–156, December 1997.

[10] Martin Burtscher, Ilya Ganusov, Sandra J. Jackson, Jian Ke, Paruj Ratanaworabhan, and Nana B. Sam. The VPC trace-compression algorithms. *IEEE Transactions on Computers*, 54(11):1329–1344, November 2005.

[11] Chao-Rui Chang, Jan-Jan Wu, and Pangfeng Liu. An empirical study on memory sharing of virtual machines for server consolidation. In *Proceedings of the Ninth IEEE International Symposium on Parallel and Distributed Processing with Applications Workshops*, IEEE ISPA, pages 244–249, Los Alamitos, CA, USA, May 2011. IEEE Computer Society Press.

[12] Jui-Hao Chiang, Han-Lin Li, and Tzi-cker Chiueh. Introspection-based memory de-duplication and migration. In *Proceedings of the 9th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, ACM VEE, pages 51–61, New York, NY, USA, March 2013. ACM.

[13] Russell Coker. Bonnie++. `http://www.coker.com.au/bonnie++/`. Accessed: 2013-10-05.

[14] Peter J. Denning. The working set model for program behavior. *Communications of the ACM*, 11(5):323–333, May 1968.

[15] Agner Fog. Instruction tables lists of instruction latencies, throughputs and micro-operation break-downs for Intel, AMD and VIA CPUs, April 2013.

[16] Rich Geldreich. miniz - single c source file deflate/inflate compression library. `http://code.google.com/p/miniz/wiki/miniz_performance_comparison_v110`. Accessed: 2013-10-05.

[17] Emilien Girault. Volatilitux: Physical memory analysis of linux systems, December 2010.

[18] Diwaker Gupta, Sangmin Lee, Michael Vrable, Stefan Savage, Alex C. Snoeren, George Varghese, Geoffrey M. Voelker, and Amin Vahdat. Difference engine: Harnessing memory redundancy in virtual machines. *Communications of the ACM*, 53(10):85–93, October 2010.

[19] Clint Huffman. Memory combining in windows 8 and windows server 2012, November 2012.

[20] Tzu-Han Hung and Alex Wauck. Towards a fully multithreading support for qemu, January 2010.

[21] IBM Corporation. *How to Improve the Performance of Linux on z/VM with Execute-In-Place Technology*, 2004.

[22] IBM Corporation. *z/VM: Performance*, 2008.

[23] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 2: Instruction Set Reference, A-Z*, 2013.

[24] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3: System Programming Guide*, 2013.

[25] Philipp Kern. Generalizing memory deduplication for native applications, sandboxes and virtual machines. Diploma thesis, System Architecture Group, Karlsruhe Institute of Technology (KIT), Germany, April 2013.

[26] Jacob Faber Kloster, Jesper Kristensen, and Arne Mejlholm. Determining the use of interdomain shareable pages using kernel introspection. Technical report, Department of Computer Science, Aalborg University, January 2007.

[27] Horacio A. Lagar-Cavilla, Joseph A. Whitney, Adin Scannell, Philip Patchin, Stephen M. Rumble, Eyal De Lara, Michael Brudno, and Mahadev Satyanarayanan. Snowflock: Rapid virtual machine cloning for cloud computing. In *Proceedings of the 4th ACM European Conference on Computer Systems*, ACM EUROSYS, pages 1–12, New York, NY, USA, April 2009. ACM.

[28] Kevin P. Lawton. Bochs: A portable PC emulator for Unix/X. *Linux Journal*, 1996(29es), September 1996.

[29] Henry M. Levy and Peter H. Lipman. Virtual memory management in the VAX/VMS operating system. *Computer*, 15(3):35–41, March 1982.

[30] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay J. Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ACM PLDI, pages 190–200, New York, NY, USA, June 2005. ACM.

[31] Peter S. Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hållberg, Johan Högberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. Simics: A full system simulation platform. *Computer*, 35(2):50–58, February 2002.

[32] Nagapramod Mandagere, Pin Zhou, Mark A Smith, and Sandeep Uttamchandani. Demystifying data deduplication. In *Proceedings of the ACM/IFIP/USENIX Middleware '08 Conference Companion*, ACM/IFIP/USENIX Middleware Conference, pages 12–17, New York, NY, USA, December 2008. ACM.

[33] Phoronix Media. Openbenchmarking.org - database test suite. `http://openbenchmarking.org/suite/pts/database`. Accessed: 2013-10-05.

[34] Phoronix Media. Phoronix test suite - an automated, open-source testing framework. `http://www.phoronix-test-suite.com/`. Accessed: 2013-10-05.

[35] Konrad Miller, Fabian Franz, Marc Rittinghaus, Marius Hillenbrand, and Frank Bellosa. XLH: More effective memory deduplication scanners through cross-layer hints. In *Proceedings of the 2013 USENIX Annual Technical Conference*, USENIX ATC, pages 279–290, Berkeley, CA, USA, June 2013. USENIX Association.

[36] Grzegorz Miłós, Derek G. Murray, Steven Hand, and Michael A. Fetterman. Satori: Enlightened page sharing. In *Proceedings of the 2009 USENIX Annual Technical Conference*, USENIX ATC, pages 1–14, Berkeley, CA, USA, June 2009. USENIX Association.

[37] Marius Monton, Antoni Portero, Marc Moreno, Borja Martinez, and Jordi Carrabina. Mixed sw/systemc soc emulation framework. In *Proceedings of the 2007 IEEE International Symposium on Industrial Electronics*, IEEE ISIE, pages 2338–2341, Los Alamitos, CA, USA, June 2007. IEEE Computer Society Press.

[38] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. *ACM SIGPLAN Notices*, 42(6):89–100, June 2007.

[39] Avadh Patel, Furat Afram, and Kanad Ghose. MARSS-x86: A QEMU-based micro-architectural and systems simulator for x86 multicore processors. In *Proceedings of the 1st International QEMU Users' Forum*, International QEMU Users' Forum, pages 29–30, March 2011.

[40] Mark Probst. Dynamic binary translation. In *UKUUG Linux Developers' Conference 2002*, UKUUG Linux Developers' Conference, July 2002.

[41] Marc Rittinghaus. Runtime benefits of memory deduplication. Diploma thesis, System Architecture Group, Karlsruhe Institute of Technology (KIT), Germany, July 2012.

[42] Marc Rittinghaus, Konrad Miller, Marius Hillenbrand, and Frank Bellosa. Simuboost: Scalable parallelization of functional system simulation. In *Proceedings of the 11th International Workshop on Dynamic Analysis (WODA 2013)*, WODA, March 2013.

[43] Mark E. Russinovich and David A. Solomon. *Windows Internals: Including Windows Server 2008 and Windows Vista*. Microsoft Press, Redmond, WA, USA, 5th edition, 2009.

[44] Prateek Sharma and Purushottam Kulkarni. Singleton: System-wide page deduplication in virtual environments. In *Proceedings of the 21st International Symposium on High-Performance Parallel and Distributed Computing*, ACM HPDC, pages 15–26, New York, NY, USA, June 2012. ACM.

[45] Abraham Silberschatz, Peter B. Galvin, and Greg Gagne. *Operating System Concepts*. Wiley, Hoboken, NJ, USA, 7th edition, 2005.

[46] Steven Sinofsky. Reducing runtime memory in windows 8, October 2011.

[47] Standard Performance Evaluation Corporation (SPEC). Spec cpu2006 benchmark description. `http://www.spec.org/cpu2006/`. Accessed: 2013-10-05.

[48] Kuniyasu Suzaki, Kengo Iijima, Toshiki Yagi, and Cyrille Artho. Memory deduplication as a threat to the guest OS. In *Proceedings of the Fourth European Workshop on System Security*, ACM EUROSEC, pages 1:1–1:6, New York, NY, USA, April 2011. ACM.

[49] Vasily Tarasov, Amar Mudrankit, Will Buik, Philip Shilane, Geoff Kuenning, and Erez Zadok. Generating realistic datasets for deduplication analysis. In *Proceedings of the 2012 USENIX Annual Technical Conference*, USENIX ATC, pages 261–272, Berkeley, CA, USA, June 2012. USENIX Association.

[50] David Ung and Cristina Cifuentes. Machine-adaptable dynamic binary translation. *ACM SIGPLAN Notices*, 35(7):41–51, July 2000.

[51] Michael Vrable, Justin Ma, Jay Chen, David Moore, Erik Vandekieft, Alex C. Snoeren, Geoffrey M. Voelker, and Stefan Savage. Scalability, fidelity, and containment in the potemkin virtual honeyfarm. In *Proceedings*

of the twentieth ACM Symposium on Operating Systems Principles, ACM SOSP, pages 148–162, New York, NY, USA, October 2005. ACM.

[52] Carl A. Waldspurger. Memory resource management in VMware ESX server. *ACM SIGOPS Operating Systems Review*, 36(SI):181–194, December 2002.