Karlsruher Institut für Technologie

Runtime Benefits of Memory Deduplication

Diploma Thesis
by

cand. inform. Marc Rittinghaus

at the Department of Computer Science

Supervisor:	Prof. Dr. Frank Bellosa
Supervising Research Assistant:	Dipl.-Inform. Konrad Miller

Day of completion: 05. July 2012

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.

Karlsruhe, July 5th 2012

Deutsche Zusammenfassung

Speicherduplikation entsteht, wenn mehrere Speicherseiten (*page frames*) den gleichen Inhalt tragen und dadurch unnötige Redundanz im Hauptspeicher entsteht. Eine empirische Studie von Speicherduplikation auf Virtualisierungssystemen hat ergeben, dass, abhängig vom eingesetzten Betriebssystem und den ausgeführten Anwendungen, der Anteil von redundanten Speicherseiten zwischen 11% und 86% liegt [15]. Weitere Untersuchungen belegen diese Ergebnisse und nennen Werte zwischen 40% [37] und 50% [22].

Um Speicherduplikation entgegenzuwirken und eine effizientere Nutzung des Hauptspeichers zu ermöglichen, sind eine Vielzahl von Deduplizierungsmechanismen entwickelt worden. Ihnen gemein ist das grundlegende Prinzip, unnötige Kopien einer Speicherseite für andere Zwecke freizugeben und stattdessen Referenzen im System über *Copy-On-Write* auf die verbleibende Instanz zeigen zu lassen. Unterscheidungen gibt es dagegen in der Art und Weise wie der jeweilige Algorithmus Duplikate identifiziert und welche Typen von Speicherseiten designbedingt erfasst werden können (z.B. nur Speicherseiten, die für I/O verwendet werden). Abhängig vom Algorithmus ergibt sich durch das Vergleichen von Speicherseiten und dem Management entsprechender Datenstrukturen ein beträchtlicher Rechenaufwand, der im ungünstigsten Fall einen vollständigen Prozessorkern auslasten kann [7]. Auch birgt Deduplizierung die Gefahr unnötiger Kosten, wenn das Zusammenlegen identischer Seiten aufgrund anschließender Schreibzugriffe rückgängig gemacht werden muss. Es ist daher entscheidend, ein besseres Verständnis über die Charakteristika von identischen Seiten zu erlangen und dieses in bestehende Algorithmen einfließen zu lassen. Auf diese Weise können Anstrengungen zur Deduplizierung auf vielversprechende Speicherbereiche fokussiert und Unkosten reduziert werden.

Frühere Forschungsarbeit auf dem Gebiet der Speicherduplikation [10, 15, 29, 38] lässt jedoch eine Menge Fragen bzgl. der Eigenschaften von identischen Seiten offen. So wurde bisher wenig Forschung betrieben, um die zeitliche und räumliche Verteilung von identischen Seiten zu erfassen und diese Daten mit Zugriffsmustern und der Art der Speichernutzung zu korrelieren. Darüber hinaus liegen keine Erkenntnisse über die Auswirkungen von Speicherdeduplizierung auf die Geschwindigkeit von Hardwarekomponenten wie Prozessor-Caches vor. Ein weiteres Problem früherer Forschungsarbeiten ist, dass aufgrund der Schwierigkeit, kontinuierliche Daten zu sammeln, die Auswertung in jedem Fall auf abgetasteten Daten beruht. Dies ist inhärent mit der Gefahr unvollständiger Informationen und daraus entstehenden falschen oder ungenauen Aussagen verbunden.

Die vorliegende Arbeit befasst sich deshalb mit der Entwicklung eines flexiblen Werkzeugs zur feingranularen Analyse von identischen Seiten sowie zur Messung von Effekten der Speicherdeduplizierung. Um die zeitliche Vollständigkeit der zugrundeliegenden Datenbasis zu garantieren, wird dabei auf die umfassende Simulation des zu untersuchenden Systems (*Full System Simulation*) gesetzt, die eine deutlich genauere Analyse erlaubt, als es in früheren Forschungsarbeiten möglich war. Darüber hinaus stehen durch den Einsatz von Betriebssystem-Introspektion (d.h. dem Aufzeichnen von relevanten Operationen im Betriebssystem) erstmals gleichzeitig detaillierte Informationen zum Systemzustand (z.B. aktuell laufender Prozess, Layout von Adressräumen, Nutzung von Speicherseiten) zur Verfügung, die eine genaue semantische Korrelation erlauben.

Eine erste prototypische Evaluation hat gezeigt, dass mit Hilfe der entwickelten Analysemethode ein genauer Einblick in die Charakteristika von identischen Speicherseiten ermöglicht wird. Neben der Bestätigung früherer Forschungsergebnisse, konnten ebenso bereits neue Erkenntnisse, z.B. über die Lebenszeit und die Effekte von Speicherduplikation auf CPU-Caches, gewonnen werden.

Acknowledgments

First of all I'd like to thank the members of the System Architecture Group for their support during my work on this thesis. This especially includes my supervisor Konrad Miller from whom I learned a lot and who always was on hand with help and advice[1]. I'd also like to thank Prof. Bellosa for his time and valuable feedback and Marius Hillenbrand who supplied computational resources and shared his comments.

My thanks go also to my fellow student and good friend, Simon Sturm, with whom I was able to master the easy and the difficult phases of the studies at the KIT.

I deeply thank my parents that they have always encouraged and supported me in pursuing my interests and that they gave me a good start into life. You are the best parents in the world!

My greatest thanks, however, belongs to my beloved girlfriend, Anna Gatzki. I'm very deeply grateful that life brought us together and I don't want to spend a single day on earth without having you by my side. Kuciu!

[1]Although I still believe that writing a framework was a good idea!

Contents

Chapter 1

Introduction

Main memory is still a scarce resource in computer systems that at the same time heavily determines the system's performance and overall scalability with regard to the accommodation of process working sets and operating system file and object caches. This is especially the case for memory intensive workloads such as database applications and virtualization-based server consolidation.

Memory duplication occurs when multiple page frames in the system's physical memory hold identical contents and thereby introduce data redundancy. An empirical study on memory duplication in virtualization hosts found the amount of redundant pages to be between 11% and as high as 86% depending on the operating system and workload [15]. Further research seconds this results and identified identical page frames in the range between 40% [37] and 50% [22]. For NUMA systems, it can be beneficial to have certain page frames duplicated to avoid frequent remote memory accesses. However, this is not the case for a single shared memory architecture on which this work focuses on. To date, a diverse set of memory deduplication techniques have been developed that aim at reducing this redundancy by copy-on-write remapping pages with equal contents to a single copy in RAM. However, deduplicating sharing opportunities that are subject to frequent modification can defeat the benefits of memory deduplication by introducing computational overhead due to early breaking of COW pages. Moreover, depending on the deduplication mechanism the detection of sharing opportunities itself is already a costly operation, which can occupy up to a whole processor core [7].

Having a sound knowledge of the characteristics of sharing opportunities is, thus, very important and helps to improve existing memory deduplication mechanisms by focusing deduplication efforts and avoiding unnecessary computational overhead. However, previous work leaves many open questions regarding the properties of identical pages [10, 15, 29, 38]. Only little research has been done on investigating temporal as well as spatial characteristics and there is no information if these correlate with access patterns and/or page usage. Moreover, we have no knowledge of the implications of memory deduplication on the system, including the performance of affected hardware components such as processor caches. Another problem with previous research is that, due to the difficulty to gather continuous data, in every case the evaluation is based on sampled data. In some cases even only a single memory snapshot of a workload is taken as data basis [10] or the measurement interval is decreased in periods of high processor demand [29]. This inherently entails the risk of distorted results.

The objective of this work is to deliver a tool chain that is capable to answer open questions regarding the characteristics of sharing opportunities as well as regarding the implications on hardware components and that guarantees the temporal completeness of its data basis to avoid imprecise results.

To achieve this goal the work at hand makes use of full system simulation and extensive continuous tracing. This way, a broad set of information about sharing opportunities in the evaluated system's physical memory are gathered and subsequently correlated to operating system state information such as executing processes, address spaces areas and page frame usage. Moreover, the full system simulation facilitates the analysis of effects of memory duplication on hardware components. A flexible scripting-based analysis interface provides the means to investigate recorded data in any way desired.

The effectiveness of the proposed analysis mechanism for sharing opportunities is demonstrated through an evaluation of the temporal and spatial characteristics of a set of prototypical desktop oriented workloads. In this course, the evaluation confirms some of the results of previous research. File cache pages for instance could be verified as a valuable target for memory deduplication as they dominate the amount of mergeable pages with up to 70% and are characterized by a high lifetime in that 65% of sharing opportunities within the file cache survive longer than 30 minutes. However, the evaluation also leads to various contradicting observations. Due to the low measurement resolution, previous work for example greatly underestimates the amount of extremely short-lived sharing opportunities. In fact, around 45% of sharings only exist for under a single second and 85% of sharings vanish before they reach a lifetime of 30 seconds. Previous research estimated the latter to be only 24.5% [38]. Further discrepancies are related to recent findings of Barker et al. who determined most sharing potential to originate from sharing opportunities within a single system and not between multiple systems (e.g., on a virtualization host) [10]. Our evaluation shows the contrary. For the first time, the proposed analysis mechanism also allows getting an insight into the sharing potential within processor caches and the evaluation reveals that large caches such as the 20 MiB L3 cache of the Intel Xeon E52470 can save over 4% of cache lines with memory deduplication.

The tool chain provided by this work helps to quickly complement previous research with additional high resolution measurements and fine granular analyses that for the first time allow a full correlation with operating system state information, thereby improving the understanding of sharing opportunities. The tracing-based architecture of our analysis mechanism, moreover, facilitates the exchange of gathered data within the research community and makes it easier for external researchers to comprehend and build on previous results.

Following the introduction, Chapter 2 provides background on memory duplication and presents related work in the field of memory deduplication. The chapter is supplemented with an overview of processor cache designs and an introduction to full system simulation with Wind River Simics. Chapter 3 discusses the weaknesses of previous research on the characteristics of sharing opportunities and weights up the pros and cons of possible data acquisition methods. Chapter 4 subsequently presents our proposed analysis method for sharing opportunities based on full system simulation. To prove the effectiveness of our approach, an evaluation is performed in Chapter 5. This chapter also sheds light on the sharing potential in processor caches. The work closes with a conclusion and prospect of future work in Chapter 6.

Chapter 2

Background

This chapter gives a short explanation on memory duplication as well as on the concepts and methods that are used to deduplicate memory. The explanations are supplemented by an overview of related work and evaluation results in the respective areas. The chapter closes with an introduction to the full system simulator Simics.

2.1 Memory Duplication

Main memory is still a scarce resource in computer systems. The amount of physical memory heavily determines the system's performance. Additional memory can be leveraged to cache frequently accessed data (e.g., file caches) and makes room for running more applications without forcing the operating system to swap. Thus, the amount of main memory effectively limits the system's scalability. This is especially true for, but not limited to, virtualization where main memory is a determining factor with regard to the number of virtual machines (VMs) that can be co-located on a physical machine.

The physical memory in modern paging-based architectures is evenly divided into small chunks called *page frames*. *Memory duplication* occurs if multiple page frames hold identical contents. Therefore, memory duplication introduces unnecessary redundancy by consuming memory, which, if freed, could be used to increase the system's speed and to accommodate the working set of more processes or larger caches[1].

[1] In systems with non-uniform memory access (NUMA), a certain degree of memory duplication can be desired to avoid remote memory accesses. This work focuses on systems with a single shared main memory.

Memory Consumption

Gupta et al. measured the amount of duplicated memory across three virtual machines (configured with 512 MiB RAM each) running Windows XP, Debian and Slackware Linux and found that nearly 50% of pages could be saved through memory deduplication [22]. Other experiments conducted with two virtual machines (512 MiB, Ubuntu Linux) compiling the Linux kernel and in another benchmark serving websites with Apache, showed a theoretical maximum saving of 40% due to redundant page frames [37]. An empirical study on memory duplication of virtual machines found the amount of redundant pages to be between 11% and as high as 86% depending on the operating system (Windows, Linux) and workload (MPI, Hadoop, MySQL, Web App) [15]. However, recent measurements indicate that over two-thirds (67%) of memory duplication may possibly stem from identical pages *within* the virtual machines and not from redundant pages across multiple virtual machines [10].

Sources of Duplication

Common sources of memory duplication are zero pages or other specifically initialized memory regions (e.g., heap), statically or dynamically loaded libraries and memory mapped files in general. Barker et al., inter alia, measured the number of identical page frames in Ubuntu Linux 10.10 while running a typical set of desktop applications (Firefox, OpenOffice, email client, etc.). They showed that over 50% of identical page frames stem from process heaps and identified library pages to be the second largest source with 43%. Especially complex GUI applications such as Firefox (48%) and OpenOffice (38%) are substantially involved in memory duplication while headless applications such as bash (9%) and ssh (6%) tend to be less critical. In total, between 5 MiB and 165 MiB of physical memory were consumed by memory duplication inside a single VM, depending on the evaluated workload and operating system [10]. A similar analysis that targeted memory duplication between multiple virtual machines running various server-type workloads (Linux kernel build, Dbench, database benchmarks) found the file system page cache (69%) and kernel pages (19%) to be the most significant causes for identical page frames [29]. Note that both studies did not account for duplicate pages that were already prevented by common operating system mechanisms such as the shared mapping of libraries. In addition, the measurements explicitly ignored zero pages as these are regularly created by the operating system for fast serving of future memory allocations as well as for security purposes when page frames are exchanged between processes. Experiments also showed that zero pages are not well suited for memory deduplication. Although there can be 20 times as much duplication from zero pages as from nonzero pages, zero pages are modified frequently which can negatively impact system performance if they are shared [38].

2.2 Memory Deduplication

Memory deduplication (in this work also referred to as simply *deduplication*) is the process of preventing or identifying and removing redundancy in physical memory by mapping virtual pages with identical contents to a single copy in RAM. Any previously mapped page frames that hold the same contents can then be freed and used for other purposes. Deduplication is therefore primarily used to reduce the physical memory usage for data-intensive workloads that tend to produce many identical page frames. To prohibit processes from modifying shared pages (and thus other processes' address spaces), the protection bits in the established mappings are set to read-only access. When a thread tries to modify a deduplicated page, the CPU triggers a page fault and traps into the operating system. If the original protection of the page allows modification, copy-on-write (COW) is employed to break the sharing and to map a private copy of the page into the process's address space. The new mapping then grants write access so that normal program execution can continue. The merging and unmerging of pages in the course of memory deduplication is completely transparent to user-mode code.

The basic memory deduplication mechanisms are mostly the same for all implementations (i.e., finding redundancy and using some form of COW), but there are also more sophisticated designs. Gupta et al. presented memory deduplication for virtual machines that in addition to full-page sharing on the basis of COW, employs page patching and page compression to share memory on a sub-page level [22]. Besides that, the actual methods of preventing or identifying and removing redundancy mainly differ depending on the targeted workload and the information available at the level of integration. The following sections give a brief overview of previous work in the area of memory deduplication and introduce the variety of algorithms developed to date.

2.2.1 Traditional Mechanisms

The major commodity operating systems (Windows, Linux and MacOS X) already employ methods to prevent memory duplication in the course of normal system operation.

File-based Methods

One mechanism is the prevention of memory duplication through shared code. Most applications depend on a set of libraries that are loaded by all applications that utilize the same APIs. Examples are system libraries such as the image loader (`ntdll.dll` on Windows, `ld.so` on Linux), programming language and subsystem specific libraries such as the C library (`msvcrt.dll`, `libc.so`) as well as libraries that are shipped as part of frameworks (e.g., .Net, GTK and Qt). The amount of memory consumed by shared code and data can be substantial. This is especially true for operating systems with a high degree of interdependence between libraries. On a Windows 7 (64-bit) even the fairly simple text editor Notepad maps roughly 80 MiB of images and system files [43]. Without deduplicating static program code and data, physical memory depletes rapidly. Therefore, the operating system detects if pages belonging to a certain library or executable are already in main memory so that further references can be mapped to existing page frames while process isolation is ensured through copy-on-write. This way, memory duplication caused by the concurrent use of shared code can be efficiently avoided.

Closely related to the sharing of executable code is the deduplication of memory mapped files. Since the same basic characteristics adhere to this source of memory duplication, the same principles are applicable. In fact, it is a common design to handle deduplication for files and executables in the same way with the help of a global file cache that does not explicitly distinguish between different types of files [13, 44].

Address Space Cloning

On some operating systems copy-on-write also prevents unnecessary duplication of page frames during process creation. While on Windows and OpenVMS new processes start with an empty address space, the parent process's address space is cloned on UNIX-like operating systems such as Linux and MacOS X [13, 20, 44].

Limitations

A major drawback of the traditional mechanisms is that they are designed around identical objects (i.e., libraries, files, etc.) but not around equal contents as fundamental unit of sharing. For that reason, these mechanisms generally offer limited applicability to workloads whose primary source of memory duplication are not shared objects but identical data in distinct objects[2] or anonymous memory regions. This limitation surfaces in particular on virtual machine hosts. While the host operating system has knowledge of the address space structure for processes running directly on the host, it cannot infer memory usage for those executed in the context of a virtual machine. Instead, the virtual machine's physical memory has to be treated by the host as anonymous memory. The semantic gap between host and virtual machine undermines the basis of traditional mechanisms that depend on the knowledge of objects and object usage. This, in consequence, leads to memory duplication between multiple VMs as well as between VMs and the host.

Traditional mechanisms for memory deduplication can also be applied to virtual machines if the concepts are properly extended. *SnowFlock* is a research project that aims at providing rapid cloning of VMs by applying the principle of traditional process forking to virtual machines. Although deduplication was not the project's motivation, the proposed method is able to reduce memory duplication through copy-on-write at the level of virtual machines [31]. A similar technique called *delta virtualization* is instrumented in the Potemkin honeypot farm to increase the virtual machine density [54]. However, a problem with these approaches is that they are only capable of avoiding redundancy for page frames that do not get modified since their time of cloning. This includes exchanging the contents of a page frame with data that is only read. The following sections present deduplication techniques that are better suited to exploit duplication even in the case of page frame modification.

2.2.2 Paravirtualization

Paravirtualization is a technique that can bridge the semantic gap between the virtual machine and the virtualization host. In paravirtualized environments the virtual machine monitor provides interfaces which go beyond

[2]Identical files stored at different locations on disk are typically handled as distinct objects because they are referenced through dedicated file control blocks.

that of a strictly virtualized computer system and which enable the virtual machine to communicate with the underlying hypervisor via *hypercalls* or simplify as well as instrument the architectural interface in order to improve virtualization performance and scalability [56]. An example of paravirtualization is *Collaborative Memory Management (CMM)*, which implements exchange of page usage and residency information between virtual machines and host to improve paging performance [45].

Disco

Disco is a hypervisor which offers a specially optimized DMA-based virtual disk interface that intercepts disk requests from the virtual machines [14, 21]. If requested disk blocks are already in physical memory (e.g., due to a previous request by another virtual machine), Disco can finish the disk request without going to disk. If the size of the disk request is a multiple of the machine's page size, the DMA operation is completed by mapping the existing page frames into the virtual machine's physical memory. This way, Disco reduces memory duplication originating from repeated disk accesses on virtualization hosts.

Satori

A comparable approach is implemented in *Satori* which focused on efficiently detecting short-lived memory duplication [38]. As in Disco, the virtual disk subsystem is instrumented to immediately detect sharing opportunities when pages are loaded from background storage. In addition, VMs provide a *repayment FIFO* to the virtual machine monitor that contains pages that can be leveraged when sharing is broken due to modification. Satori was implemented for the Xen hypervisor [9]. Naturally, it performs best on disk-intensive workloads. Experiments with httperf [40] and Apache showed that up to 94% of total sharing opportunities could be exploited. In contrast, Satori found to be less suited for workloads that dynamically create duplication in memory. During a Linux kernel build only between 18% and 50% of memory duplication could be addressed.

I/O Deduplication

Koller and Rangaswami focused on the deduplication of I/O requests that deliver identical disk blocks [30]. To achieve this effect, a content-based disk cache is employed that uses a combination of sector and content-hash-based lookup depending on the type of request (read or write). A content-based cache can further reduce memory duplication because identical data from different sectors is not duplicated in memory as it is the case with Disco and Satori.

2.2.3 Memory Scanners

Memory scanners take a different approach towards finding sharing opportunities than solutions based on paravirtualization. The mechanisms presented so far are solely based on capturing memory duplication which stems from repeated access to identical data in background storage. As Satori illustrated, this method performs less well if the degree of dynamically created content is high. Memory scanners in turn are able to exploit these sharing opportunities. They can also be used in environments where paravirtualization is no option because the target system cannot be modified (e.g., due to certification issues or the lack of source code).

The basic idea behind memory scanners is to periodically scan the physical memory to locate identical page frames. When redundant page frames are identified, the scanner frees them by remapping the referencing pages to one remaining copy. To accomplish this task, the operating system needs to support reverse mapping to enable the scanner to find the address spaces or virtual machines that own the pages of interest. This is a drawback of memory scanners because it is common that operating systems do not offer reverse mapping at all or only for specific types of pages (e.g., only private pages but not shared pages or anonymous pages but not file-backed pages) [13, 44]. Another disadvantage of memory scanners is that one must trade off the introduced overhead against the rate at which pages are scanned and new sharing opportunities are located.

VMware ESX

Waldspurger presented memory deduplication based on memory scanning for VMware ESX [55]. The implementation uses page frame hashing to identify pages with potentially identical contents. The hashes are recorded in a hash table and a lookup is done to locate sharing candidates. After a successful match, a full comparison of the page contents follows to verify that the pages are identical and that they can be merged. Sharing metrics from production deployments of ESX Server showed that between 7% (of 10% maximum sharing potential) and 33% (of 43%) of total pages could be reclaimed.

Kernel SamePage Merging (KSM)

In contrast to the virtual machine focused implementation in VMware ESX, the *Kernel SamePage Merging (KSM)* daemon is a memory scanner built into the Linux kernel which scans pages that are specifically advised as mergeable, no matter if they belong to a process or a virtual machine [7]. However, because of architectural constraints in the Linux kernel (e.g., incomplete reverse mapping), KSM is only able to deduplicate anonymous memory regions. In virtualization environments KSM is often used in conjunction with QEMU [11] and its kernel counterpart KVM [28]. To benefit from memory deduplication, QEMU marks the anonymous memory regions that represent virtual machine physical memory as mergeable. Unlike the ESX memory scanner, KSM uses multiple red-black trees to find suitable sharing candidates, whereas the pages' contents itself is utilized as node keys. Depending on the workload, the latter negatively influences merging performance as page modifications can easily lead to a considerable degeneration of the red-black trees [37].

Miller et al. substantially improved the performance of KSM by using a hash table to find redundant pages and by adding a hinting mechanism that introduces a prioritized processing of I/O generated memory duplication [37]. The presented approach combines the notion of I/O guided memory deduplication as found in Satori and Disco with memory scanners, but without the dependence on paravirtualization.

2.2.4 Hardware Solutions

A drawback of all software-based solutions to memory deduplication is the computational overhead they inevitably introduce. Difference Engine shows a performance overhead within 7% of the baseline [22]. Satori has a worst-case overhead of 34.8% for chunked reads (according to the authors, the impact is negligible for random reads) [38]. Although memory scanners allow to trade off overhead against deduplication performance through the adjustment of the scan rate, they generally come with worse overhead characteristics. Arcangeli, Eidus, and Wright report a constant single core CPU usage of up to 10% for a recommended setting in a KSM/KVM environment (approx. 47MB/sec, Intel Q9300 2.5 GHz) [7].

Hardware-based approaches are an alternative to software-based solutions. HICAMP implements a novel content-addressable memory architecture that is designed to ease multi-threaded programming by transferring synchronization work into the hardware layer and by providing native support for programming language and OS structures such as threads, iterators, read-only access and atomic updates [16]. Due to its content-addressability-based design, HICAMP also inherently offers transparent memory deduplication. Evaluations showed memory savings in the range from 1.5x to more than 4x over conventional memory for text data such HTML web pages and JavaScript scripts. For the VMmark virtualization benchmark [53] a memory footprint compaction of the VMs by a factor between 1.86x and 10.87x was measured, while ideal page-based memory deduplication only ranged between 1.44x and 5.21x. However, for binary data with high entropy (e.g., compressed JPEG) a 10% overhead in memory usage was found due to negligible savings through compaction that could not balance out the additional management overhead.

2.3 Processor Caches

This work also analysis the effect of memory deduplication on processor caches. This section provides background on the cache hierarchies of two popular x86 processors: the Intel Pentium 4 single-core processor (Willamette) [27] and the Intel Core-i7 quad-core processor (Sandy-Bridge) [1]. The presented cache hierarchies are used as reference designs for the simulated caches instrumented in the evaluation to measure implications of memory duplication on cache utilization.

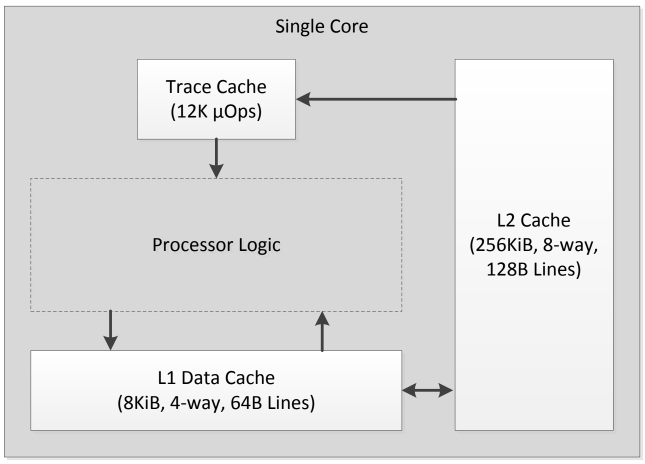

Single Core
Trace Cache (12K µOps)
Processor Logic
L2 Cache (256KiB, 8-way, 128B Lines)
L1 Data Cache (8KiB, 4-way, 64B Lines)

Figure 2.1: Simplified Intel Pentium 4 Cache Hierarchy

The Pentium 4 single-core processor with Willamette core was released on November 20, 2000. Although performance and feature-wise outdated, the Pentium 4 still offers a good target platform for single-core processor simulation. Figure 2.1 shows a simplified version of the processor's cache hierarchy. The CPU owns three on-chip caches organized in two levels. The trace cache is positioned between the instruction decoder and the actual processor logic (register banks, ALUs, FPUs, etc.) and can hold up to 12.000 micro-operations. Data is cached in a dedicated 4-way set associative level 1 data cache with a total capacity of 8 KiB (128 lines, 64 bytes per line). Both caches are connected to an 8-way set associative level 2 cache that stores both data and instructions. The L2 cache has a total capacity of 256 KiB (2048 lines, 128 bytes per line).

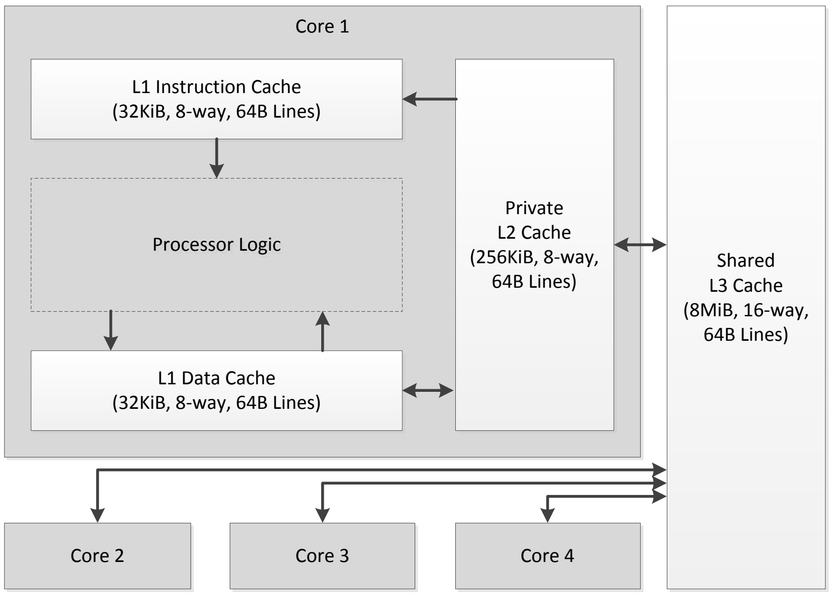


Figure 2.2: Simplified Intel Core-i7 Cache Hierarchy

The Core-i7 with Sandy-Bridge architecture is a substantially more recent processor that was released in January 2011. In contrast to the Pentium 4, the Core-i7 is a quad-core processor. The caching system internal to each core is structured similarly to the one in the Pentium 4. Notable changes are the move away from the trace cache towards a standard L1 instruction cache and the change in the caches' organization. The 8-way set associative L1 caches are each increased in size and can now hold up to 32 KiB of data or instructions respectively (512 lines, 64 bytes per line).

Although, the L2 cache's capacity stayed the same, the organization of the caches has been brought into line, so that the L2 cache in the Core-i7 now uses 64 byte lines, too. In the course of the fundamental change towards a multi-core processor, the CPU's cache hierarchy is also supplemented with a third level cache. While the L1 and the L2 caches are private to each core, the L3 cache is shared among the four cores. With 8 MiB, it provides the majority of the overall caching capacity. The L3 cache is designed as a 16-way set associative cache with a line size of 64 bytes for a total of 131072 lines.

2.4 Wind River Simics

Simics [35, 57] is a cycle-accurate full system simulator developed by the Intel subsidiary Wind River. It allows doing fully deterministic simulations of a computer system and is thus a suitable platform for operating system and device modeling as well as research on operating system and hardware behavior at the software/hardware boundary. Some of Simics key features are, in no particular order: [48]:

- High-level component-based configuration system
- Scripting environment with integrated Python 2.5 interpreter
- Save and restore of checkpoints and persistent data
- Run-time code generation for various targets (ARM, MIPS, PPC, SPARC, x86 and x86-x64)
- Run-time inspection of virtual components (CPU, memory, etc.) via hooks, callbacks and breakpoints
- Generic parameterized cache simulation
- User developed simulator extensions

To balance between accuracy and performance, system models are designed and simulated on a functional level. The inner workings of devices are for the most part only simulated as far as they are exposed to the operating system and applications in real hardware. In that sense, processor instruction set extensions (MMX, SSE, etc.) are implemented, but the modeling of timing at a detailed level such as CPU pipelines is omitted[3].

[3]Detailed models can be build with custom components [18].

Naturally, the simulation performance heavily depends on the level of accuracy (i.e., send instruction fetches to the memory hierarchy or not) and the type of inspection features enabled. Measurements on a 933 MHz Pentium III showed for an x86 target an average simulation speed in the range of 2.1 MIPS and 5.7 MIPS [35]. The 3.4 GHz Core-i7 host used in this work reached an average speed between 20 MIPS and 30 MIPS. However, activating the inspection of memory accesses drops the performance by a factor of 10. Thus, for simulating a 2 GHz Pentium 4 one must expect a slowdown at least by a factor of 1000. This does not include doing any analysis on the gathered data. Note that although multiple simulation cells (i.e., independent groups of components) can be simulated in parallel, a single virtual Pentium 4-based PC for example is still treated as a single cell and thus simulated in a single thread.

2.4.1 Virtual Platform Configuration

In Simics a virtual platform (i.e., one or more connected simulated machines) is configured with the help of a scripting-based configuration language. The language itself supports common structured programming constructs such as `if(-else)`-clauses and various types of loops (`for`, `while`, `foreach`). The language also tightly integrates Python for more advanced scripting tasks. In fact, all commands available in the Simics command line interface (CLI) are implemented as Python functions [4].

```
create-motherboard-440bx $motherboard acpi = TRUE
        rtc_time = "2012-05-07 13:15:00 UTC"
        bios = "rombios-2.65.2.11"

create-processor-pentium-4 $cpu freq_mhz = 20 cpi = 1
connect $motherboard.cpu[0] $cpu.socket

create-simple-memory-module $dimm0 memory_megs = 2048
connect $motherboard.dimm[0] $dimm0.mem_bus

create-std-ide-disk $hdd size = 8589934592 file = "d0.craff"
$motherboard.southbridge.connect ide0_master $hdd

instantiate-components
```

Listing 2.1: Simics Script. The script creates primary components of a x86 PC and connects and instantiates them.

A virtual system is assembled from user-created *components*, each representing a piece of hardware which connects to other parts of the system through standardized interfaces [39, 49]. The definition of components typically follows closely the structure in a real system. Listing 2.1 illustrates how an exemplary virtual system is configured. The system consists of an Intel 440BX-based motherboard, a 20[4] MHz clocked Pentium 4 processor, 2 GiB main memory and an 8 GiB IDE hard disk drive. The disk's data is contained in a specifically formatted image file. Typical additional components are a VGA graphics adapter, a console with connected mouse and keyboard and a network interface controller. It is also possible to create user-defined components by writing Python or C-based extensions.

The final step in the configuration is the instantiation of the components which is comparable to the instantiation of classes in object-oriented programming languages. After that, the configured components are accessible as objects via the CLI and the Simics API for simulator extensions. The system's configuration can also be extended at run-time.

2.4.2 Run-Time Inspection

Simics offers three major entry points for run-time inspection of simulated components [49]:

Object Attributes The easiest way to get an insight into the state of components is to read the attributes they expose. A processor object for example offers access to all its registers, the FPU state, the CPUID values, pending interrupts and exceptions and many more. In most cases, the configuration settings are also available (e.g., the processor frequency). In addition, some components do statistics which they publish with the help of attributes (e.g., the number of committed CPU instructions).

Haps (Callbacks) Callbacks (in Simics referred to as *haps*) are the primary method for run-time inspection that goes beyond reading object attributes. A callback is a user-defined function (in a Python script or a C-based simulator extension) that is registered to be called by Simics at the

[4]A small processor frequency is recommended for interactivity because the virtual system's clock is bound to it. In the case of 20 MHz, the virtual clock advances one second for every 20 million instructions executed.

occurrence of a certain user-chosen event. Simics offers many generic events such as control register read and write, instruction execution or processor's privilege level change. Other events are target specific such as x86-TLB related ones. Heavily used in this work are the *Magic Instruction Event*, that is triggered whenever the CPU executes a special instruction (`xchg bx,bx` for the x86-target), and the *Periodic Event*, which is periodically executed after a user-chosen amount of CPU cycles have passed.

Interfaces Interfaces are very similar to callbacks in that the user can declare functions that are called during the simulation. But in contrast to callbacks, interfaces cannot stand alone but must be implemented by user-defined components. These components can in turn be used to connect to or intercept the communication of other components that support the supplied interface. Example interfaces are the *Timing Model Interface* and the *Snoop Memory Interface* that allow to plug into the memory hierarchy of the simulated system to observe, delay and to some degree alter memory transactions before and after they are submitted.

2.4.3 Cache Simulation

The processor models shipped with Simics do not emulate a cache hierarchy as this would only unnecessarily slow down the simulation. However, if the behavior of the CPU's cache hierarchy is of interest, Simics provides the means to do explicit cache simulation. As mentioned in Section 2.4.2, Simics offers the possibility to monitor memory transactions that flow through the system's memory hierarchy by implementing a special interface. Therefore, cache simulation in Simics is done with the help of a dedicated cache component that implements the respective interfaces to plug into the memory hierarchy. Figure 2.3 shows an exemplary setup with a simplified memory hierarchy and a two level cache hierarchy connected to it.

The memory hierarchy is modeled by Simics regardless if a cache simulation is done and is used to control the flow of memory transactions in the virtual system. As illustrated in the figure, the view of the memory depends on the position the respective device connects to the memory hierarchy. The processor for example has access to all memory ranges, including the ones that do not map to the virtual RAM but to APIC control registers or GPU memory. The denoted DMA device accesses the virtual RAM's memory space instead and therefore does not see additional map-

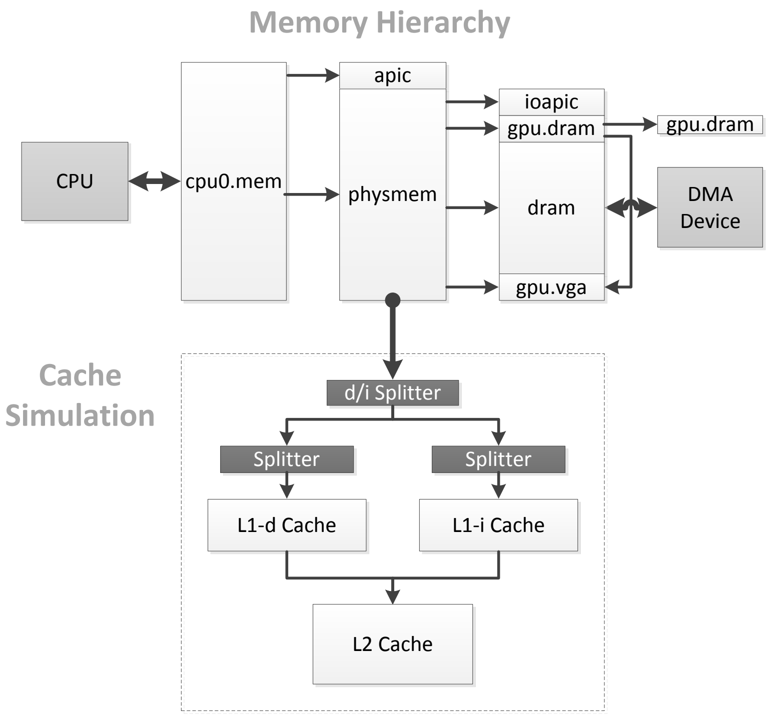


Figure 2.3: Simplified Memory Hierarchy with Cache Simulation. The cache is attached to the `physmem` memory space and observes CPU memory transactions but not device ones. The cache is modeled as a hierarchy of generic parameterized cache components.

pings. Observing the memory transactions at `physmem`, however does not show transactions submitted by the device as these are not visible at this layer [34].

The cache hierarchy is built from multiple interconnected instances of a generic user-configurable cache component, called *g-cache* [5]. The most important parameters of a g-cache are:

- Number of Cache Lines
- Cache Line Size
- Associativity
- Replacement Policy
- Virtual/Physical Tagging
- Read/Write Timing Penalty

The caches are not directly connected to the memory hierarchy but use a set of memory transaction splitters to prepare the stream of memory transactions before they reach the caches. The i/d splitter is an instruction/data splitter that separates memory transactions caused by instruction fetches from those caused by regular data accesses as part of normal code execution. In the exemplary cache hierarchy in Figure 2.3 the separated streams are then targeted at different g-caches which thereby mimic a caching system with distinct data and instruction caches. The splitters in front of each cache additionally split memory transactions that cross cache line boundaries as it is the case in a real processor.

The g-cache itself does not hold any contents but only models the assignment of cache lines. If the cache is forced to write-back data or fetch a line, a pseudo memory transaction is generated that is send to any component that listens on the cache (i.e., a next level cache or an analyzer component). Since the cache simulation is orthogonal to the memory hierarchy, transactions generated by a cache will not be visible to observers of the memory hierarchy but only to those that listen on the respective cache.

To do an analysis of a cache's behavior, one can choose from a number of statistics that each cache does (hit-rate, number of fetches, etc.). This includes the inspection of the cache lines. As already mentioned, it is also possible to analyze the memory transactions performed by a cache.

Chapter 3

Analysis

The effectiveness of memory-deduplication techniques can be improved if information about the characteristics of sharing opportunities (i.e., a group of pages with identical contents) is incorporated. Miller et al. substantially improved the sharing performance of KSM by prioritizing the scanning of pages that stem from I/O [37] as these have been found to be an important source of memory duplication [29, 10]. Especially, memory scanners can benefit from a sound knowledge of the statistical properties that duplicated pages show. Identifying potential "hot-spots" that tend to develop an increased amount of memory duplication can improve the effectiveness of scanners by focusing their search on promising pages. Concentrating memory deduplication efforts to interesting areas also increases the efficiency through the reduction of wasted computational overhead. Besides *finding* sharing candidates it is important to *trade off* possible memory savings against potential performance losses that are caused if established sharings need to be broken at a later time because of page writes. Satori showed that the number of sharing opportunities originating from zero pages is approximately 20 times greater than from non-zero pages [38]. However, since zero-pages are used by the operating system to provide memory for allocations, it is probable that, if these are shared, many zero-pages need to be broken at once when the newly allocated memory is used. This adds overhead at an unfavorable moment and possibly at time where the system experiences memory pressure. Thus, the question arises if certain types of sharing opportunities should be ignored if performance is crucial.

The next section gives an overview of the shortcomings of previous evaluations. Section 3.2 then discusses which method is appropriate to gather the data required for a thorough understanding of sharing opportunities.

3.1 Analyzing Sharing Opportunities

Previous work that deals with memory duplication and mechanisms for deduplication leaves many open questions regarding the characteristics of sharing opportunities:

- Only little research has been done to study the types of pages involved in sharing opportunities and the evaluations do not state if sharing opportunities can typically be assigned to a single type of page or if they tend to span multiple categories. In addition, solely Barker et al. distinguish between identical pages within a single virtual machine (*Self-Sharing*) and those that span multiple OS instances (*Inter-VM-Sharing*).

- Satori is the only work that did an analysis on the temporal characteristics of sharing opportunities. However, the results only differentiate between zero-pages and non-zero pages. The research that focused on page types did not investigate the temporal characteristics and how these can be correlated.

- There is no information available if sharing opportunities in general as well as in respect to the types of pages they include show a characteristic spatial distribution.

- With the exception of measuring the number of pages that can be freed, no work evaluated the effects that deduplicating memory has on the applications and the system as a whole. This includes implications for the efficiency of hardware such as processor caches.

- Previous work is limited in the temporal resolution of the analysis. Analyzing sharing opportunities is methodically difficult to do, as the entire system needs to be stopped to search for and classify identical page frames. For that reason, previous research resorted to an evaluation based on sampled data. For Satori an interval of thirty seconds was chosen to take memory dumps from virtual machines. Barker et al. based their evaluation on weeklong real-world traces that contained memory dumps for every thirty minutes. For the supplementing benchmarks they took a single memory snapshot per VM after the test workload had been executed. Due to the way their memory deduplication algorithm works, Kloster, Kristensen, and Mejlholm only measured sharing opportunities during idle periods.

As stated before, having detailed knowledge of the characteristics of sharing opportunities can substantially improve the efficiency of deduplication efforts. Therefore, a method is required that allows to easily capture and analyze all the information that is needed to get a thorough understanding of the types of sharing opportunities that deduplication techniques have to face.

To overcome the shortcomings that previous analyses showed, the following areas need to be covered:

Temporal Characteristics Having information on the temporal characteristics comprises the ability to make statements about when memory duplication occurs (e.g., during phases of increased I/O) and how long groups of identical pages exist. This enables memory deduplication techniques to be triggered by events that are known to cause much duplication. Thus, overhead during less interesting phases can be reduced. I/O guided deduplication mechanisms are already a step into this direction. However, not taking the average lifetime of sharing opportunities into account can as well hurt performance through wasted deduplication efforts on too short-lived opportunities. For some types it might also be more efficient to consider the *temporal development* and to delay any deduplication so that the contents of involved pages can stabilize.

Spatial Characteristics Similar to the temporal properties, data on the spatial characteristics of identical pages are of great value for deduplication mechanisms. The latency of memory scanners depends on the configured scan-rate and on the spatial distribution of duplicated pages. Those which form contiguous regions can be deduplicated faster than those that spread over the whole memory. In the case of comparable workload (e.g., cluster of web servers), deduplicating the physical memory of virtual machines might also benefit from heuristics that first try to find equal (inter-VM) pages at similar locations. Another important metric is the typical rank of sharing opportunities (i.e., the number of pages with identical contents). It helps focusing deduplication on promising areas.

Access Characteristics Comparable to the L-shape policy used in various power management systems [50], an analysis should investigate if there is a relationship between memory access patterns and page stability. If similar policies like the L-shape policy can be applied to memory, timing of deduplication efforts can be made more efficient.

Page Usage A fundamental property of pages is the type of usage they are subject to (e.g., file-backed page, heap page, anonymous page, etc.). Since this information can easily be gathered online, heuristics based on page usage are potentially a good basis for deduplication decision algorithms. Therefore, an analysis should correlate data on access patterns and temporal as well as spatial characteristics with data on page usage.

Implications Deduplicating memory has diverse effects on the system's operation. Memory allocations are handled differently as memory is freed (which might be reclaimed at a later time). This leads to a different layout of pages in physical memory. An increased degree of memory sharing plays into the hands of processor caches and the TLB but potentially harms performance on NUMA systems. This effect should be considered when memory deduplication is done. Thus, information on the implications of memory deduplication needs to be gathered. This work focuses on the implications on the processor's cache hierarchy.

3.2 Data Acquisition Method

To be able to gather all the information required for a thorough analysis as described in Section 3.1 restricts the set of suitable methods by which the data is collected. The method needs to measure access patterns and temporal properties with a high granularity while it must not be too intrusive to distort evaluation results. Moreover, it requires access to system-wide memory as well as data structures to identify page usage and to include pages that are used internally to the operating system only.

The following sections give an overview of commonly used techniques for data acquisition and discuss their advantages and disadvantages with regard to the analysis of sharing opportunities. For the rest of this work, the term *target system* or simply *target* refers to the entire operating system or application that is to be examined.

3.2.1 Performance Counters

Modern processors provide *performance counters* as a mean to do run-time CPU performance monitoring [25]. They enable software to measure the number of times configured hardware internal events occurred and thereby permit to capture information on run-time behavior otherwise transparent to application code. Measurable events are for example the number of retired instructions or the number of mispredicted branches, whereas the set of supported counters is specific to each processor model.

Since CPUs often also provide access to cache related counters (e.g., number of cache misses) they are suitable for a basic performance analysis of the processor's cache hierarchy. This is necessary for the evaluation of memory deduplication effects. However, a fundamental problem with performance counters with regard to the requirements of this work is that they only provide the means to count events but not to inspect them. They do not give information on the contents or characteristics of operations (e.g., addresses, type of memory accesses, etc.). Another challenge in the analysis of sharing opportunities in a running system is that because of multi-tasking and asynchronous device operations, the state of the system, such as the contents of memory pages, can change at any time. Thus, the operating system needs to be modified to enable the analysis code to run without preemption. This allows synchronously scanning all system memory and finding identical pages. However, it is not sufficient if page access characteristics need to be measured as the analysis code is not capable to track individual memory accesses. So in practice additional changes to the operating system are required. Furthermore, resorting to a "stop-and-scan" method inherently leads to a sample-based analysis which does not meet the granularity requirements of this work.

Performance counters are thus not a suitable method for the analysis of sharing opportunities and the effects of memory deduplication. Especially, because they require multiple intrusive changes to the behavior of the target system which leads to distorted results. This is particularly problematic with regard to a cache performance analysis.

3.2.2 Dynamic Recompilation

Another conceivable approach to the analysis of sharing opportunities is the *dynamic recompilation* of the target. The concept of dynamic recompilation is based on the idea of run-time program code replacement. The new code semantically executes the same logic but can be instrumented in required ways. In the context of this work, this might be additional code to track each memory access that the original program code executes. *Valgrind* is a programming tool which uses dynamic recompilation techniques to do memory debugging, memory leak detection and profiling [51].

An advantage of this approach is that the target's program code does not need to be modified on a source level to allow analysis code to run. This fully preserves the original program logic of the target and thus avoids deviating measurement results. In contrast to performance counters, dynamic recompilation is also not reliant on a sample-based analysis since operations can be tracked at the moment they are executed. Therefore, a fine granular analysis of the temporal and spatial characteristics of sharing opportunities is feasible. Scanning memory pages can be done "between the lines" of the target program code. Since all memory accesses are observable, the analysis of effects on the cache hierarchy is possible, too. *Gleipnir* is a memory profiling tool based on the Valgrind framework [19] that records memory accesses. The traces can then be used in conjunction with the cache simulator *DineroIV* [17]. This way, effects on the cache hierarchy are made visible through a dedicated cache simulation run after the target system was executed. It is also possible to perform the cache simulation on memory accesses at run-time. In general, doing a cache simulation instead of monitoring the real processor's cache hierarchy has the benefit that arbitrary statistics can be generated. It is also possible to examine different cache organizations based on the same dataset.

A problem with dynamic recompilation as implemented in Valgrind though is the restriction of the recompilation to a single application. The analysis code can therefore only consider sharing opportunities within the address space of a single running process. Expanding the instrumentation across all applications would introduce massive synchronization overhead due to multi-tasking and it would still leave out pages which are not used by user-mode code. Thus, dynamic recompilation would also need to cover the operating system kernel to allow a complete analysis of sharing opportunities. However, this is very difficult to realize and may not be feasible for all kernel code paths. Furthermore, expanding dynamic recompilation onto the whole system is a very intrusive approach and therefore not well suited for this work. Even restricted to a single application, dynamic recompilation is not fully non-intrusive. Although the original program logic is preserved, the environment in which this logic executes is altered. The additional program code and data originating from the dynamic recompilation consumes memory in the address space of the target process. Thus, if the process is very memory intensive, it may hit memory quotas or experiences an exhaustion of virtual address space. Moreover, the timing conditions in the communication with other processes are influenced when dynamic recompilation is employed.

3.2.3 Virtualization

Virtualization[1] is used by most previous work (e.g., [10, 29, 37, 38]). This is due to the fact that the technology is widely available and the virtual machines' physical memory is a promising target for memory deduplication and therefore a realistic use case. Moreover, virtualization makes it easy to monitor one or more entire systems. In contrast to the already mentioned data acquisition methods, analyzing sharing opportunities in a virtualized environment benefits form the fact that the code performing the analysis can run outside of the target system. Consequently, the measurements regarding sharing opportunities are guaranteed not to be distorted by the analysis method itself. In addition, the hypervisor can pause the virtual machines at any time. This makes it substantially easier to get a consistent view of the memory (to find duplicated pages) than it is possible for analysis code that runs within the target system.

However, moving the analysis out of the system introduces a semantic gap between the analyzer and the target. For instance, the information on page usage can no longer be inferred. Instead, paravirtualization techniques or other communication channels are required to make certain virtual machine internal information available for code running directly on the virtualization host. Depending on the overhead of such a mechanism, measurings can potentially get distorted. Another weak point is that, in contrast to binary recompilation, virtualization offers no easy way to track individual memory accesses. Therefore, monitoring the cache hierarchy is to some extent only possible with the supplementary use of performance counters. This restricts virtualization to a sample-based analysis since the target needs to be periodically paused to track sharing opportunities.

Regarding the requirements of this work, virtualization is comparable to the use of performance counters. Although virtualization offers a much

[1]In this work, the term *virtualization* always refers to virtualization based on *trap-and-emulate* [47] or hardware extensions such as *Intel-VT* [26, 42] or *AMD-V* [6], in which cases unmodified target code is run.

better platform for memory analysis, it lacks the tracking of individual memory accesses and thereby reduces the evaluation resolution and limits the scope of a cache performance analysis.

3.2.4 Custom Hardware

It is also possible to take a hardware-based approach to analyzing sharing opportunities. Specialized RAM modules are a feasible approach [8] as they allow fully observing any memory activity as well as monitoring the contents of page frames. Another approach is the development of a specialized processor [41]. This way, sharing opportunities can be found and continuously tracked in hardware. In contrast to virtualization or performance counters, these methods do not need to resort to a sample-based examination and are therefore suited for a fine granular analysis of temporal and spatial characteristics. However, like virtualization, a hardware-based approach needs extra logic to overcome the semantic gap between the analyzer component and the target system. Otherwise, no information on page usage can be gathered. A possible solution is to write page usage information in a defined area in physical memory so that they are accessible to the hardware.

With regard to the examination of deduplication effects on the caching infrastructure, a custom processor offers great potential. It can incorporate a specialized cache that is capable of measuring interesting memory duplication related statistics at run-time and which publishes these with the help of a dedicated interface (e.g., a special CPU instruction). A hardware-based solution that is implemented in the RAM modules though would need to resort to a cache simulation. This could be done directly in hardware or in a subsequent simulation step in software (e.g., with DineroIV). However, the latter depends on the recording of all memory accesses which leads to huge amounts of data if no compression is applied. Doing a run-time compression in turn increases the hardware complexity.

In sum, specialized hardware is conceptually able to fulfill the requirements of this work. A major drawback, however, is the high development complexity. This is especially the case for such central components like a custom processor. Depending on the type of hardware customized (e.g., CPU), compatibility with commodity operating systems and applications may also be compromised. This questions the representativeness of evaluations done on such a platform. Another inherent problem is the limited flexibility bound to hardware-based solutions. While it is easy to change analyzer software to examine new aspects, hardware designs typically demand greater efforts to be adapted to new requirements.

3.2.5 Full System Simulation

Full system simulation emulates an entire physical machine. In contrast to virtualization, it does not depend on hardware support or concepts such as trap-and-emulate. Instead, it uses binary recompilation or *binary translation* [3] to run the target code in a confined virtual environment. Binary translation is an advanced form of binary recompilation that is capable to run target code compiled for a different instruction set architecture than the host system provides. This is done by a run-time translation of the foreign machine instructions into instructions available on the host system.

Since an entire physical machine is simulated, full system simulation is not restricted to the context of a single application. All code of the target is covered by the translation/recompilation process. This eliminates a major drawback of the previously mentioned application-level recompilation. With the full instrumentation of all applications, operating system and drivers, a complete and continuous analysis of sharing opportunities in the simulated machine is feasible. Every memory access can be inspected by an analyzer component. As it is the case with virtualization, the analyzer resides outside the simulation and thereby preserves exact measurement results. However, full system simulation suffers the same way from the semantic gap between the simulator and the simulated system. Thus, additional logic is needed to communicate target internal information.

Another advantage of using simulation is the fully deterministic behavior of the target system. This is especially useful, if the same experiment is intended be run several times, each time with slightly altered parameters. In the context of this work, this may be a comparison between the cache performance with and without activated memory deduplication. Full system simulation also makes it very easy to analyze the implications of memory deduplication on the hardware. Since all components in the target are simulated, the virtual hardware can be customized to gather relevant information on-the-fly. This makes simulation substantially more flexible than specialized hardware. The flexibility, though, comes with a high computational overhead (see Section 2.4). Hence, full system simulation is potentially the slowest of the presented data acquisition methods.

3.3 Conclusion

Having a sound knowledge of the characteristics of sharing opportunities helps to improve existing memory deduplication techniques by focusing deduplication efforts and avoiding unnecessary computational overhead. However, previous work leaves many open questions regarding the properties of identical pages. Only little research has been done on investigating temporal as well as spatial characteristics and there is no information if these correlate with access patterns and/or page usage. Moreover, we have no knowledge of the implications of memory deduplication on the system, including the performance of affected hardware components. Therefore, a method is required that is capable of doing a thorough analysis of sharing opportunities and the effects of deduplication.

Areas of interest are: temporal, spatial and access characteristics, page usage and deduplication effects. To cover these areas, a suitable data acquisition method needs to be employed. Table 3.1 summarizes the advantages and disadvantages of the presented methods:

	Perf. Counters	Dynamic Recomp.	Virtua-lization	Custom Hardware	Full Sys. Simulation
Continuous	○	●	○	●	●
Synchronous[1]	◐	●	●	◐	●
Complete[2]	◐	○	●	●	●
Non-Intrusive	○	◐	●	●	●
Deterministic	○	●	○	○	●
Flexible	◐	●	●	○	●
Fast	●	○	●	●	○
Cache Eval.	◐	●	◐	●	●

[1] Analysis runs synchronous with the target and thus has a consistent view of its state.
[2] Method is capable to analyze sharing opportunities in the whole target.

Table 3.1: Comparison of Data Acquisition Methods. Full system simulation offers the best trade off.

Compared to the other data acquisition methods, full system simulation offers the best trade off. It combines the advantages of dynamic recompilation and virtualization in that it allows doing a semantically, temporally and spatially complete analysis of sharing opportunities. At the same time it profits from the abstraction of virtual machines with regard to analysis complexity and intrusiveness. As pure software solution it is also more widely available and can easily be adapted to new data acquisition and evaluation requirements. For these reasons, this work builds, in contrast to most previous work, on **full system simulation** as the data acquisition and evaluation environment.

Chapter 4

Design and Implementation

Full system simulators only deliver the basic platform on which the target system can be run. To analyze sharing opportunities within the system the simulator needs to be complemented with additional software. The required components can be divided into two main areas: *data acquisition* and *data analysis*. The software components responsible for the acquisition collect the data (i.e., sharing opportunities, memory accesses, page usage information, etc.), which in a second step is used by the analyzer components to compute statistics related to the characteristics of sharing opportunities. With the help of these software components, a fine granular analysis can be accomplished.

The first section starts with an overview of the fundamental considerations that were laid out as basis for the design of the software components. The following sections introduce the design of the acquisition and analyzer software; describe how each of the required information can be retrieved from a full system simulator and how this data can be efficiently correlated and analyzed.

4.1 Design Considerations

Choosing full system simulation as data acquisition method only defines the environment into which additional software needs to be embedded. To get a comprehensive solution for the examination of sharing opportunities requires each software component to be carefully designed and directed towards the goals of this work. Hence, a set of design goals have been defined that software components should meet:

Completeness A fundamental requirement for the software is that it is capable to measure all information necessary to overcome the shortcomings of previous work (see Section 3.1). This includes closing the semantic gap between the simulator and the simulated system to gather data internal to the target (e.g., page usage).

Completeness also requires that the software does not resort to a sample-based inspection, but instead gets invoked on crucial events (e.g., memory accesses) so that each type of information is complete with regard to its temporal development.

In addition, completeness aims at the ability to analyze the collected data in any way necessary to extract desired results.

Intrusiveness Measuring and analyzing phenomena within the system they occur, is always bound to the risk that measurements get distorted by the observation. This is especially true for the examination of sensitive components such as processor caches. As illustrated in the previous chapter, full system simulation does a great step towards a non-intrusive analysis as it allows additional software components to reside outside the target system. However, since this work depends on target internal data, minor additions to the target have to be made to communicate this information to the components outside the system. Thus, an important design goal is to keep the introduced overhead as minimal as possible.

Efficiency As showed in Section 2.4, full system simulation is potentially the slowest of the presented data acquisition methods. The time needed for a single simulation pass can range from hours up to several days, depending on the amount of details simulated and the complexity of the analysis. It is therefore desirable that any additional software components work efficiently and that the design itself takes high simulation times into account. This includes the ability to do multiple analyses in one simulation pass[1].

As important as efficiency in time, is an efficient use of space. Since huge amounts of data are gathered during a single simulation, a deliberate use of storage space is just as important.

[1]The term *simulation pass* denotes the phase until the simulation reaches a point after which no further measurings need to be taken (e.g., after a benchmark run) and the simulation can be stopped.

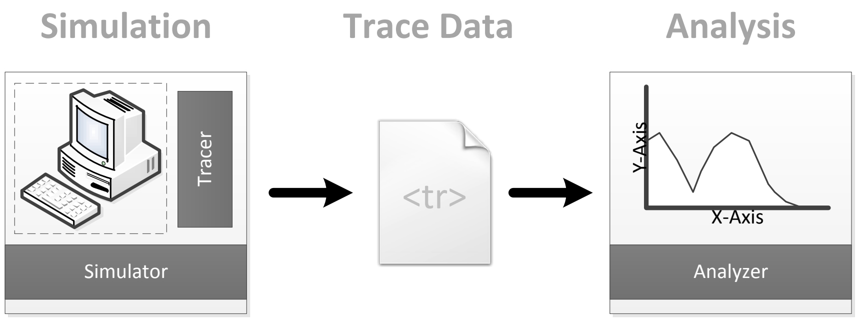


Figure 4.1: Decoupled Data Acquisition and Analysis Phase. A tracer records interesting information during a simulation and stores that data in a specially formatted file. Afterwards, dedicated software is used to analyze it.

Flexibility A last design goal regards the flexibility of the introduced software components and the general design they are based on. This comprises the ability to easily extend and adapt the software as well as the ability to run arbitrary analyses on the gathered data.

Based on these considerations, a software architecture has been chosen that decouples the data acquisition phase from the analysis. This is illustrated in Figure 4.1. While the simulation of the target system runs, a *tracer* component embedded into the simulation software gathers all required information and stores it as efficient as possible in a specially crafted trace file. After the simulation is stopped, this file can be opened with dedicated software which in turn allows executing arbitrary analyses on the collected data.

Although such a two-stage design increases the software's complexity, it pays off with regard to efficiency and flexibility. Decoupling simulation and analysis allows examining a single simulation's measurements based on different criteria without the need to rerun the simulation itself. Since running a simulation is potentially a very time-consuming process, having an independent analysis phase makes this approach substantially more efficient than an integrated solution with regard to the total required run-time (especially if many distinct analyses are planned). On a technical perspective, it also offers the possibility to parallelize computations. While the simulation is an inherently single-threaded process, it is no problem to have several analyses concurrently executing side-by-side on the same trace.

An overall reduced run-time also increases the flexibility to do a more comprehensive exploration of the gathered data. Moreover, storing the basis of any analysis (i.e., the trace data) allows working on the dataset even if the simulation environment itself is not available. The tracing-based design thereby simplifies the exchange of material with other researchers and makes it easier for others to comprehend and build on previous research results.

Completeness and a non-intrusive acquisition are properties exclusive to the tracing module and the analyzer and are therefore not affected by the presented architecture. The Sections 4.2 and 4.3 explain how these components are designed and how they meet the chosen criteria.

Simulation Plaform

The implementation of this work is based on the *Wind River Simics 4.2 (x86)* [57] full system simulator. It provides all features that are needed by the proposed tracer design. This includes the ability to write extensions for the simulator as well as to retrieve all data that is of interest to this work via public hooks and callbacks. Section 2.4 gives further information on the software.

Alternative x86 full system simulators are *QEMU* [11], *Bochs* [12] and *MPTLSim* [58]. However, QEMU and Bochs do not offer a cache simulation which is crucial to this work. In addition, not all information sources are accessible through official interfaces which make the implementation more complex. MPTLSim is an interesting alternative that, in contrast to Simics, utilizes processors models that allow a cycle accurate simulation below the instruction level. Consequently, CPU internal mechanisms such as the whole cache hierarchy are integrated, whereas Simics depends on a dedicated cache simulation. However, the flexibility that comes with a separated cache simulation is advantageous for this work, as different cache hierarchies are evaluated.

4.2 Data Acquisition

The data acquisition is encapsulated in the tracer component. Since the tracer runs as an extension of the simulator it has access to all information regarding the state of the simulated machine. Its responsibility is to inspect the machine's state at certain points in time and store this information in a trace file. This file can then be processed with the help of the analyzer software.

A system simulator is based on the concept of binary translation or binary recompilation. This allows the simulator to instrument each executed target CPU instruction. In addition, hardware operations such as memory accesses can be made visible. The simulation software exposes this instrumentation through hooks and/or callbacks that software can utilize to inspect the state of the machine and to monitor operations (e.g., memory accesses) and events (e.g., interrupts) that occur within the simulation. To get invoked, the tracer is designed to either make use of any available hooks and callbacks, which is preferred, or, if no other means are provided, it can be directly build into the corresponding code paths within the simulator. In contrast to a periodic, sample-based measurement, the tracing guarantees the temporal completeness of the information, since every event that changes the state of the simulation in a relevant form can be captured and recorded if necessary.

Due to the nature of code instrumentation, the tracer is invoked by the simulator synchronously to the simulation itself. Therefore, the state which the tracer sees is always consistent and the simulation continues as soon as the tracer finishes its state inspection. Since the internal clock of the target is decoupled from the wall clock time, the overhead introduced by the tracer has no effect on the state of the simulation. However, the time consumed by a full simulation pass may increase. This depends on the amount and detail of information that the tracer is configured to gather and the overhead that is bound to the retrieval of the respective information.

To collect and store all the information necessary to build statistics about the properties of sharing opportunities as described in Section 3.1, several information sources need to be covered by the tracer. These are described in the next sections. For each of these sources one or more *trace providers* are responsible to react to events and supply corresponding trace data.

4.2.1 Memory Inspection

To examine the characteristics of sharing opportunities, the first step is to gather information about which page frames in the simulation's physical memory hold equal contents and what temporal and spatial properties these groups of frames have. This can be accomplished with *memory inspection*. Memory inspection describes the process of inferring information about sharing opportunities by just reading the simulation's physical memory. No additional semantic information about the page frames (e.g., their type of usage) is known in this step.

Sharing opportunities can be formally represented through sets of page frames with identical contents, called *sharing groups*. The number of frames contained in a single sharing group (i.e., the number of page frames with a certain contents) determines its *rank*. A sharing group with a rank of 4 therefore denotes a group of four page frames with equal contents. Based on the definition of sharing groups, the minimal rank is 2 and there naturally can only be a single group per page contents. In sum, sharing groups are characterized by their referenced contents, their rank and the page frames they hold.

Detection of Sharing Opportunities

To make statements about sharing opportunities, the tracer first needs to identify sharing groups in the physical memory of the simulated machine. Figure 4.2 illustrates the basic logic. To get an initial list of sharing groups present in physical memory, all page frames are marked as *dirty* when the tracing of sharing opportunities is activated. Marking a frame as dirty removes the frame from its sharing group (if any), deletes the sharing group if its rank falls below 2 and prepares the frame for further processing.

For each dirty frame, the tracer first searches for an existing sharing group that references the same contents. If such a group can be located, it is expanded by adding the examined frame. This increases the group's rank by 1. If no sharing group can be found, the tracer tries to find at least one other frame with equal contents so that a new sharing group can be build. In the case this attempt also fails, the frame does not belong to a sharing opportunity. Nevertheless, each frame that has been processed is marked as clean and does not need to be visited again. Since the tracer only *reads* page frames in the course of this logic, the target's state is not affected. Hence, the algorithm is completely non-intrusive.

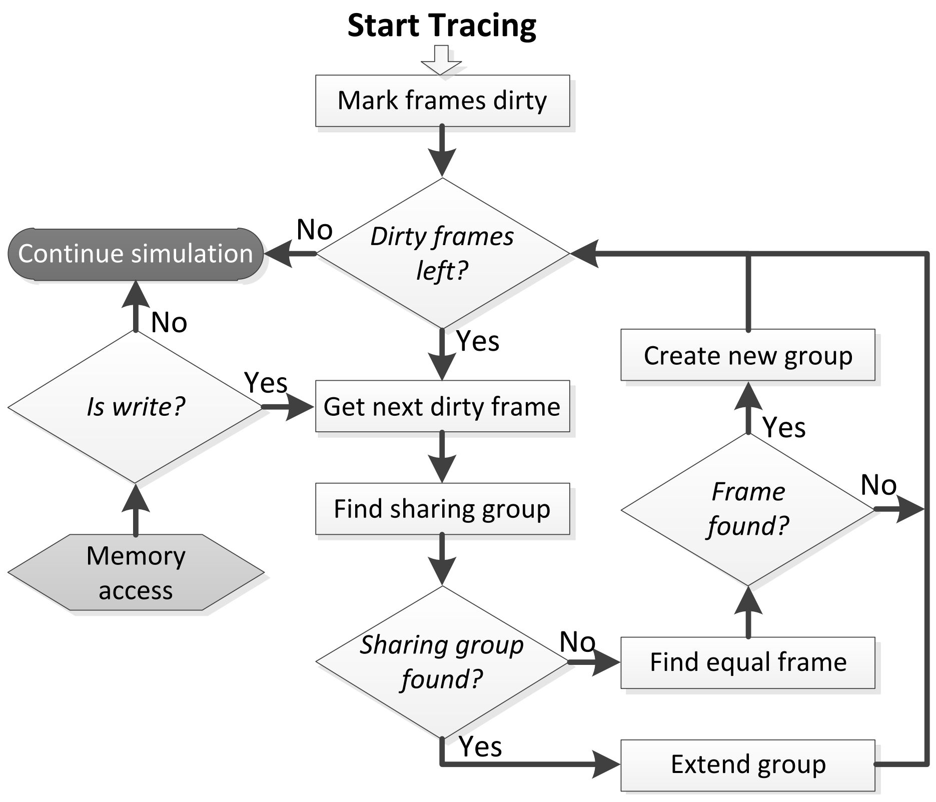


Figure 4.2: Simplified Program Flow of Memory Inspection. At the beginning an initial list of sharing groups is created. Afterwards, write accesses trigger an update of the list to reflect the contents modifications.

The list of sharing groups is only guaranteed to be valid at the time it is created. As soon as the simulation continues and the memory gets modified, existing sharing opportunities may vanish and new ones may emerge. To ensure that the sharing groups reflect these changes, the tracer inspects every subsequent memory access. While read accesses can be safely ignored, write accesses need further examination. As soon as a modification to the page frame is applied, the tracer marks the page frame at which the write operation was targeted at as dirty and starts the detection of sharing opportunities. This time it only covers the accessed page frame. This way, the sharing groups always represent the sharing opportunities present in the physical memory of the simulated machine.

However, updating the sharing groups at every write access comes at the cost of a high computational overhead and, with regard to the simulation speed, is the most expensive operation performed by the tracer. Depending on the rate at which write accesses are issued by the simulated machine, a slowdown factor of 10 is realistic. Despite optimizations in the implementation, this cost needs to be paid to allow the fine granular study of sharing opportunities.

It is important to differentiate sharing opportunities by the contents of the frames they involve. This way, sharing opportunities based on zero-pages can be statistically treated separately. To take this requirement into account, the tracer incorporates a pattern recognition that is able to mark sharing groups that reference frames with previously configured patterns.

The generated sharing groups already expose much information about the general sharing potential of the target as well as on the spatial characteristics of the detected sharing opportunities. This includes the total number of page frames that can be deduplicated and thereby the amount of memory that can potentially be freed as well as statistics about the size of sharing opportunities. The latter can be computed with the help of the sharing groups' rank. Since the individual page frames which participate in the memory duplication are known, the spatial distribution of these frames in physical memory can also be examined. However, up to this point, it is not possible to make statements about the spatial distribution in *virtual memory* as no mapping information is available.

To increase the informational value of sharing groups, they are supplemented with important timing information. This additional information adds the ability to examine basic temporal characteristics of sharing opportunities. Every time a new sharing group is created, a timestamp (*Create Time*) is added. This way, the lifetime of sharing groups can be calculated when they are destroyed. Furthermore, every time a modification to the sharing group is made, a timestamp (*Last Modified Time*) is updated to reflect the time of change; allowing calculating the amount of time a sharing group has been stable.

Implementation

As already mentioned in the previous section, the detection of sharing opportunities is a costly operation, since it has to be done after each write access. In the implementation, the detection of sharing opportunities is

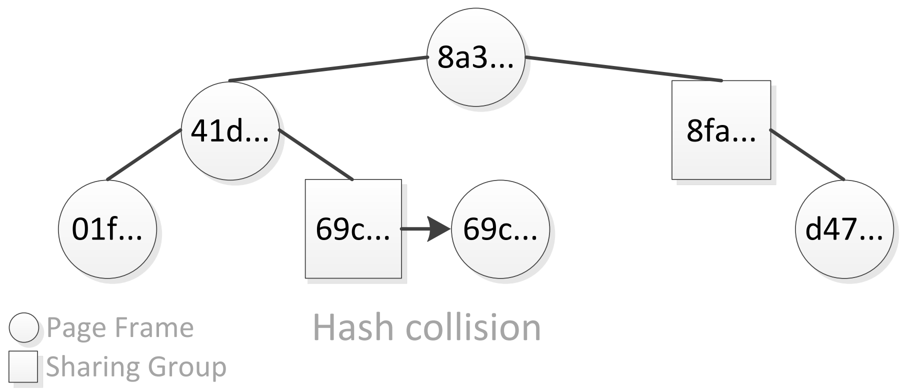


Figure 4.3: Red-Black Tree to detect Sharing Opportunities. Page frames are added to a red-black tree via their hashes. Nodes are either the frames itself or sharing groups. Hash collisions are resolved with chaining.

optimized with the help of a red-black tree and hardware accelerated CRC-32C hashing [23]. The nodes of the tree are either single page frames or sharing groups that contain multiple frames. The CRC-32C hash of the frame's contents is used as key. This way, searching a sharing group or a page frame as illustrated in Figure 4.2 is reduced to a single lookup in the tree (and a subsequent frame compare on success). If a frame with the same contents is found, the node is replaced with a sharing group. Existing sharing groups are extended. If the lookup fails, the frame is added to the tree as a new node. In addition, chaining is used to resolve hash collisions.

Tracing of Sharing Opportunities

The basic software architecture depends on the ability of the analyzer software to solely work on the data contained in the trace files. The tracer must therefore record enough data so that the analyzer software is able to reconstruct the sharing groups that were present at any time during the simulation. Since resorting to a sampled output is no option, the tracing is based on the three major events which characterize the detection of sharing opportunities:

- Creating a new sharing group
- Expanding an existing sharing group
- Detecting writes that do not result in sharings

Every write operation that the tracer inspects leads to one of these events. The tracer records enough information for each of the events to enable the analyzer software the reconstruction of the sharing groups (including all previously mentioned properties). Note that the tracing does not include the payload of the write access itself. The analyzer therefore cannot infer the exact contents of page frames. However, it is possible to configure the tracer to store full frame dumps for sharing groups that reach a specified rank.

4.2.2 Operating System Introspection

Having information on sharing groups alone does not suffice to correlate sharing opportunities with the workload that created or benefited them as information on the simulation's operating system state (e.g., running process) is missing. Hence, it is for instance not possible to examine if certain processes tend to produce more memory duplication than others or if certain types of address space areas (e.g., anonymous memory) show a particularly high amount of sharing opportunities.

Moving the tracer code outside of the target system leads to a semantic gap between the simulated machine and the tracer. This is caused by the fact that the tracer can no longer use services provided by the simulated operating system to retrieve system and application specific information (e.g., enumerate processes). Moreover, the direct access to kernel-mode data structures is substantially more complicated because the memory addresses of kernel objects are unknown. This is especially the case for dynamically created objects.

A solution to get information about the current operating system's state is to find relevant data structures through the search for known patterns or strings in the physical memory of the simulated machine. These identifiable memory locations may allow inferring the address of data structures by adding/subtracting a certain offset. However, this approach is very sensitive to changes in the operating system (e.g., new kernel version), possibly insufficient to gather all state information and might not even be possible for operating systems such as Windows that are able to page out kernel memory. Furthermore, applying this concept to a running system leads to a very time-consuming, constant polling, examination and interpretation of the relevant data structures.

Operating system introspection is capable to close the semantic gap by actively communicating the occurrence of relevant events to the tracer. This way, the tracer does not rely on constant memory polling but can get informed about changes in the operating system state when they occur. To implement the operating system introspection the respective code paths in the kernel of the target system are instrumented to collect and synchronously send information to the tracer. For a process creation this might be for instance the path of the executable image and the id of the new process. For the analysis of sharing opportunities such data is of special interest as it allows correlating the occurrence and properties of sharing opportunities with certain system activity.

The information traced by the introspection employed in this work can be divided into four fundamental areas:

- Process Management and Dispatching
- Kernel-Mode Address Space Management
- User-Mode Address Space Management
- Virtual-To-Physical Page Mapping

The first area comprises the creation, destruction and dispatching of processes and address spaces. The two following areas cover all events that lead to a structural change of the kernel or any user address space. These are for instance the allocation or removal of virtual address space areas such as file-mappings. However, this explicitly excludes individual memory allocations that do not change the address space layout (e.g., regular heap allocations). The last area refers to all information about events that change the virtual-to-physical mapping of pages.

Having all this information available in the trace data allows the analyzer software not only to reconstruct the sharing groups at every time of the simulation, but also to reconstruct all relevant parts of the operating system state. This way, the usage of a frame can be determined by resolving the address spaces and processes that use the frame at the time of interest and doing a lookup on the type of mapping that is applied for the respective virtual page. In addition, sharing information as well as for instance cache statistics can be brought into connection with state information such as the current active process or address space.

The advantage of introspection in respect to plain memory inspection is that the instrumentation code can use and access all operating system services and data structures as if the tracer was integrated directly into the simulated machine. Hence, information can be gathered in a very powerful and at the same time convenient way. An inherent problem with introspection, however, is the fact that the target needs to be modified. This is not possible for closed-source operating systems such as Microsoft Windows. Depending on the communication method and the amount of information that needs to be transported, the overhead of introspection can also quickly reach a point that is not justifiable with regard to the overall intrusiveness. It is therefore crucial to employ a suitable communication method.

Communication

A fundamental challenge of the semantic gap is to transport the information from within the target machine to the tracer that runs as extension in the simulator. One of the design goals of the tracer is to affect the state of simulation as little as possible. Standard high-level communication channels such as those provided by Ethernet or other commonly used communication interfaces (COM, USB, etc.) are not an option as these introduce too much side-effects to comply with the design goals. In most cases, they rely on the allocation of additional memory (e.g., for data packets) and the data potentially traverses multiple software or protocol layers until it is sent. Since even primitive operations such as the mapping of a page frame need to be communicated, this complexity might also lead to nested tracing of operations. This in turn increases the complexity and overhead of the communication code as it has to gracefully handle these situations. Another problem is the latency with which events are recognized by the tracer. Since the tracer timestamps events when they are received, the difference between the actual time that the event happened in the simulation and the time that the tracer recognizes the event needs to be as small and predictable as possible. Thus, a very lightweight communication channel needs to be utilized.

The binary translation used by full system simulation is well suited to build such a lightweight communication channel. Since every instruction can be instrumented, it is possible to define a seldom used "magic" instruction and to invoke a callback within the simulator each time this instruction is executed by the target. In Simics this is supported through the *Magic Instruction Callback*. The magic instruction for the x86 architecture is defined

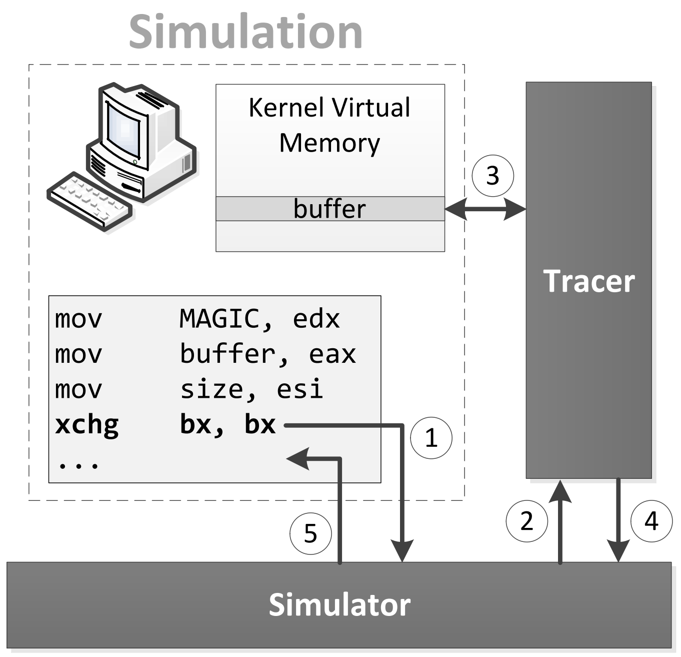


Figure 4.4: Communication based on a Magic Instruction. The target stores introspection data in a buffer, executes a magic instruction and the tracer is invoked. When the buffer is processed, the simulation continues.

as `xchg bx, bx` as this operation has no effects in the context of a regular program and, thus, minimizes the number of unintentional invocations of the simulator.

With the help of the magic instruction the data transport itself can be done with minimal simulation internal overhead. The payload is stored in a buffer in kernel virtual memory and a set of CPU registers are used to pass the buffer address and size to the magic instruction callback handler in the tracer. Since the buffer is typically very small (depending on the type of information), it can always be allocated on the kernel stack of the thread that performs the action, which is to be traced. Thus, the intrusiveness of the communication is comparable to an extra regular procedure call. Moreover, since the kernel stack of the currently running thread is always guaranteed to be present in physical memory, nested tracing operations are prevented.

To retrieve data from the payload buffer, the callback handler in the tracer needs to translate the kernel virtual address of the buffer into the corresponding simulation physical address by using the currently active page directory. Afterwards, it can read the specified amount of bytes. If the buffer crosses page frame boundaries, additional address translations are necessary.

Virtual Machine Support

Due to the high probability of identical page frames between VMs that are running the same operating system, virtualization is a common use case for memory deduplication. It is therefore desirable that the tracer as well as the analyzer software is capable to handle the analysis of virtual machines within the simulation. This includes the detection of sharing opportunities and the support for virtual machine introspection.

The Simics version used in this work does not support processor models that come with the instruction set extensions for virtualization (Intel-VT or AMD-V). For that reason, the implementation of the tracer has been directed to handle virtualization based on binary translation. However, the design is also compatible with other virtualization technologies. The current implementation is based on QEMU as the virtual machine monitor (VMM) [11]. Without the hardware-based virtualization extension KVM [28], QEMU hosts each virtual machine in a separate process. Hence, each virtual machine's physical memory is represented through a correspondingly large contiguous memory area in the address space of the respective QEMU process. Figure 4.5 depicts this setup.

The tracer needs no additional logic to detect sharing opportunities in this scenario. Since identical page frames between VMs are also equal in the physical memory of the simulated virtualization host, the presented algorithm is able to track them. The identification of the virtual machines which are referenced by a sharing group could optionally be done through the reverse mapping of the corresponding page frames to the QEMU virtual machine monitor processes that host the VMs.

Performing operating system introspection in a virtualized environment, however, requires special handling. As the physical memory of virtual machines is mapped as anonymous memory in the host, the information received through the operating system introspection in the host kernel is not sufficient to resolve the page usage of VM owned page frames. The additional level of abstraction within the simulation introduces another semantic gap. Therefore, the operating system state of the VMs is not trans-

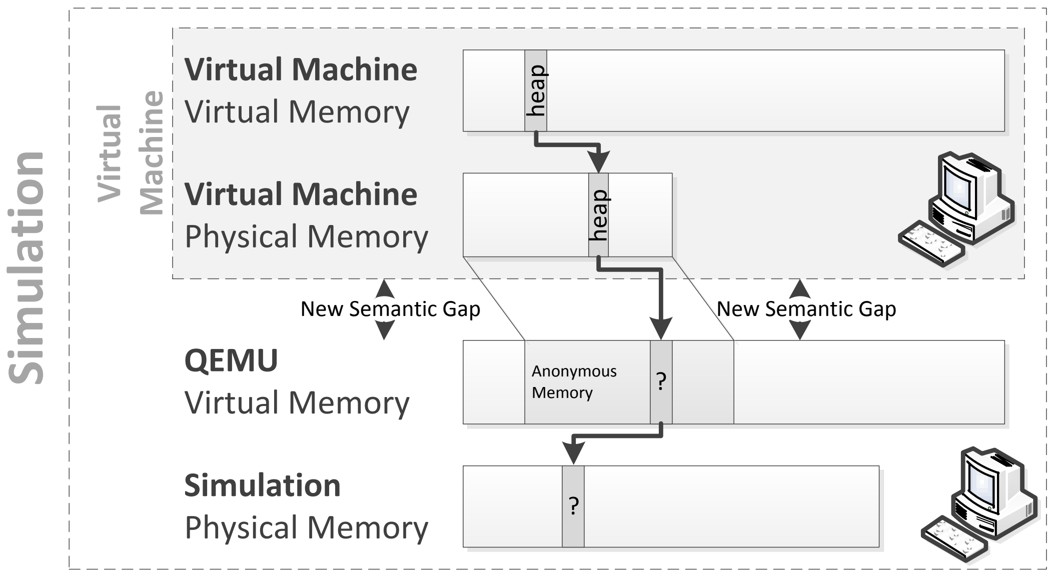


Figure 4.5: Semantic Gap. The physical memory of a simulated virtual machine is mapped as anonymous address space area. Any information about the page usage is lost.

parent to the tracer. To close the semantic gap, the introspection must also cover virtual machines. Although this can be accomplished by running an introspection-enabled kernel within the simulated VMs, additional modifications in the three following areas are required:

Communication Channel The binary translation employed by QEMU optimizes the target code by removing unnecessary instructions. Since the magic instruction defined by Simics falls into this category, the invocation of the tracer from within introspection-enabled VM kernels is unintentionally removed. A way to bypass this mechanism is to choose a magic instruction for virtualized systems that is not targeted by binary code optimizations. This instruction is then trapped by the virtual machine monitor which in turn can forward the call to the simulator through executing the original magic instruction. Figure 4.6 illustrates the additional indirection. In the implementation, the magic instruction for virtual machines is defined as a read from the model specific register (MSR) with index `0x40000000` through the `readmsr` instruction [24]. The MSR address range between `0x40000000` and `0x400000FF` is marked as a reserved range that is guaranteed not to be used by any processor in the future [26].

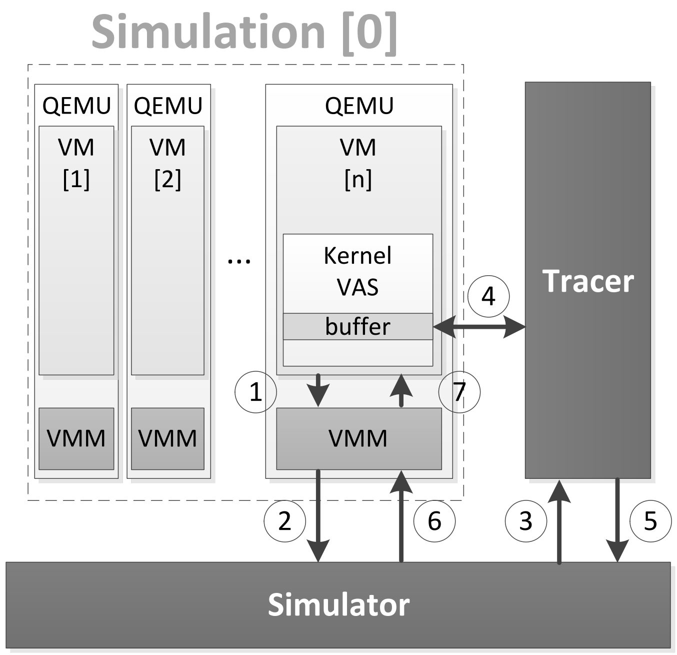


Figure 4.6: Communication with Virtualization. A system ID identifies the source of events. For virtualization with binary translation, extra steps in the communication can be necessary.

An alternative to this approach is to modify the code optimization performed by QEMU to skip the removal of the original magic instruction.

Operating System Identification As the tracer receives introspection data from multiple machines, it must be able to determine the system to which the data belongs. This can be solved by tagging each system with a unique identifier and letting it include this identifier with every invocation of the magic instruction. Therefore, the tracer requires each simulated machine (including the virtualization host) to request a system id before the first introspection data is sent. In Figure 4.6 the system id is denoted by the number between the square brackets. Since the simulated virtualization host starts before any virtual machines, its system id is always 0. Virtual machines are in turn identified by a system id greater than 0.

Multi-Level Address Translation The last extension required to support virtual machines relates to the translation of the payload buffer's address. If a virtual machine stores introspection data in a buffer, this buffer is allocated on the kernel-stack of the currently running thread. However, since this thread runs in the virtual machine, the buffer's address sent to the tracer is only valid in the virtual address space within the virtual machine and cannot be used to access the data in simulation physical memory.

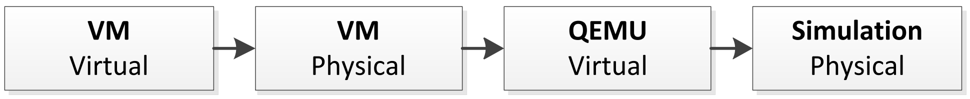


Figure 4.7: Three-Stage Address Translation. To access the payload buffer in a virtual machine, the tracer has to perform multi-level address translation.

Figure 4.7 depicts the tree-stage address translation that is applied by the tracer whenever it detects that the buffer resides in a virtual machine (using the system id). The buffer VM virtual address is first translated into a VM physical address. Subsequently, the VM physical address is translated into a virtual address pointing into the contiguous address space area in the QEMU VMM process that represents the VM's physical memory. The last step translates this virtual address into the corresponding physical address in the simulation's physical memory. This address can finally be used to read the buffer.

To accomplish this translation each invocation of the magic instruction needs to include a pointer to the currently active page directory that translates the buffer's virtual address into the physical address. Since for virtual machines the payload buffer is not accessible prior to the first complete address translation, the system id and the pointer to the page directory must be supplied via registers. Another information required by the translation process is the location of the physical memory address space areas in each QEMU VMM to enable the second translation step. In the implementation, QEMU has been modified to provide this offset at the start of a new virtual machine.

In practice, a single address translation from VM virtual to simulation physical requires between 4 and 6 nested address translations, depending on if the virtual address resides in a huge page or not. This is caused by additional address translations required to traverse the page table hierarchy of the virtual machine.

Linux Kernel Introspection

For evaluation purposes, the Linux kernel (version 3.3.2) has been extended to support operating system introspection. Table 4.1 gives an overview of the kernel operations being traced:

Process Management and Dispatching	• Process: *create/terminate/switch* • Address Space: *create/destroy*
Kernel-Mode Memory Management	• Build-Specific Kernel Layout[1] • Machine-Specific Kernel Layout[2] • Zoned Buddy Allocator: *allocate/free* • Slab Cache: *create/destroy/expand/shrink* • Large Kernel Allocation: *allocate/free* • Virtual Kernel Allocation: *link/unlink* area • Page Cache: *add/replace/remove* page • Kernel Modul: *load/unload* • Kernel Stack: *allocate/free*
User-Mode Memory Management	• Address Space Area: *create/adjust/destroy*
Virtual-To-Physical Page Mapping	• Page Directory Entry[3]: *set/clear* • Page Table Entry: *set/clear*

[1] Segments (text, data, bss), kernel start page, etc.
[2] Number of page frames, high memory start frame, etc.
[3] Only for huge pages

Table 4.1: Linux Kernel Introspection Overview

4.2.3 Cache Performance Analysis

The processor models shipped with Simics do not include a cache simulation. However, Simics provides the possibility to do a dedicated cache simulation with the help of the generic cache class *g-cache* (see Section 2.4.3 for details). The class allows building arbitrary cache hierarchies with customized cache properties (size, organization, etc.). The current implementation of the tracer uses the g-cache to model and trace various caches.

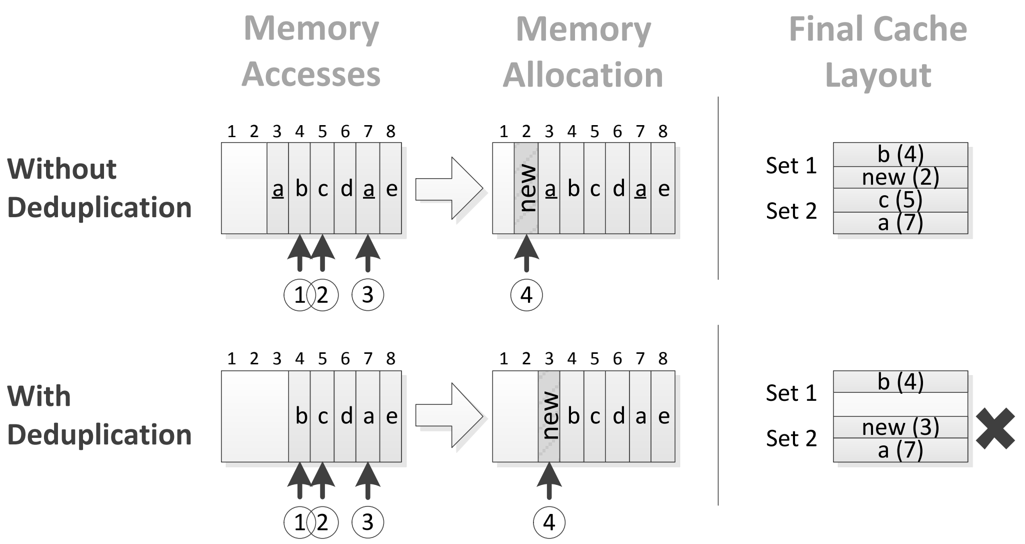


Figure 4.8: Cache Assignment with and without Memory Deduplication. Deduplicating memory changes how the operating system serves memory allocations. This has complex effects on cache line assignment.

To get an insight into the effects of memory deduplication on the processor caches, it is not possible to modify the cache simulation to emulate the effects. The implications of memory deduplication on the cache are too complex. Freeing page frames through deduplication changes the way the operating system serves memory allocations in the future. Hence, the placement of semantically same frames possibly changes with activated deduplication, leading to the replacement of otherwise preserved cache lines. Figure 4.8 illustrates this problem. A pure cache simulation cannot estimate how the operating system would use free memory. Emulating all possible behaviors quickly leads to a state explosion and is not feasible. Instead, two dedicated simulations (one with and one without activated memory deduplication) need to be traced and compared. Since the simulation is fully deterministic any changes regarding the cache utilization and performance must be caused by memory deduplication.

The g-cache exposes most of the interesting performance metrics via attributes. This way, external components can access statistics on the number of fetches, the current hit-rate and many more. To record these attributes, the tracer incorporates a trace data provider which periodically captures user-configured object attributes. The current implementation does not offer an alternative to the sample-based tracing of attribute values. However, the interval can be configured on a CPU cycle basis and therefore allows maximum precision if desired.

To supplement to the statistics supplied by the g-cache, several memory duplication related metrics have been added:

- The number of cache lines referencing mergeable page frames
- The number of referenced unique mergeable page frames
- The number of referenced unique sharing groups

Subtracting the last two values for instance gives an indication on how much the cache benefits from memory deduplication. If the number of unique mergeable frames and the number of unique sharing groups is close to equal, nearly all mergeable frames referenced by the cache are located in different sharing groups. Merging the frames within these sharing groups will therefore have only little effect on the cache.

Other information periodically captured by the tracer is the contents of the cache (i.e., the referenced page frame numbers). This allows the analyzer to find the processes for each cache line that own the respective page frame. This data can be used to make statements about the importance of scheduling policies for the cache utilization in relation to memory deduplication. It might be for instance beneficial to co-schedule processes which own page frames in the same sharing group.

4.2.4 Data Organization and Storage

Tracing all the information presented in the last sections quickly leads to the accumulation of huge amounts of tracing data. A billion records per simulated minute is a realistic rate for a simulated single-core 20 MHz processor. This demands for an efficient storage mechanism to reduce the size of the resultant trace file. Otherwise, traces can quickly grow up to several TiB in size and get overly complicated to handle. Moreover, depending on the performance of the storage backend, the rate at which new trace data needs to be written can saturate the I/O bandwidth, thereby slowing down the simulation. The implemented storage system overcomes these challenges through a trace data compression that saves up to 99% of storage space. At the same time, it structures the trace data in a way that enables the analyzer software to quickly extract the data that it is looking for. To accomplish this the tracer organizes trace data into the following basic primitives:

Entries Each datum that the tracer generates is packed into a structure called *trace-entry*. An entry is the smallest unit of information and every data that is to be stored in the trace file needs to be encoded into one or more of such trace entries. The size of a single trace-entry is 10 bytes, regardless of the data it holds. The format of a trace-entry, i.e., how these 10 bytes need to interpreted, is only partially defined and apart from that chosen by the respective data provider that generates the entries. Hence, the format of a trace-entry containing information on a certain operating system introspection related event is for the most part completely different from a trace-entry describing a CPU memory access. The part which is equally formatted for both entry types is the first (header) byte of a trace-entry. It encodes the fundamental type of the entry (e.g., introspection data, memory access, metadata, etc.). The header also specifies if this entry is a continuation of the previous one. This way, arbitrary long trace information can be stored by chaining multiple trace entries into a single large one.

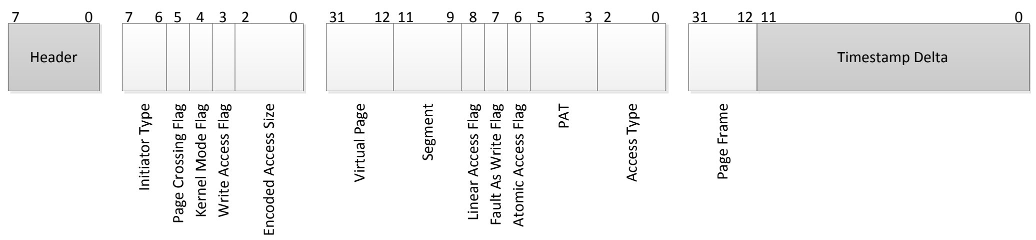


Figure 4.9: Entry for a Data Memory Access

Figure 4.9 exemplary depicts the format of a trace-entry encoding a CPU memory access. The last 12 bits of the entry represent the number of elapsed CPU cycles since the last entry generated by the data provider. The time stamping is, however, not fixed to this format and can also be expressed with the help of a specially formatted metadata entry that is chained to the actual entry. Thereby, a 64 bit absolute timestamp can be used if the format of the trace-entry or the value of the timestamp delta requires it.

Streams To organize semantically connected trace entries, the tracer uses the abstraction of *streams*. For each source of information which is to be traced, a dedicated stream is registered by the respective trace provider. For each stream, the tracer's stream management code returns a unique *stream handle* that allows the corresponding trace provider to store a trace-entry in a specific stream. The operating system introspection data for each system (i.e., the simulation itself and simulated virtual machines) is for example stored in a dedicated stream. The same is true for the information on sharing groups as well as for each cache attribute that is traced. This way, streams enable the tracer and the analyzer software to distinguish between different data sources.

The tracer also registers some internal streams that do not hold trace information. For instance, the tracer allows each stream to be associated with arbitrary description information (expressed via key/value pairs) such as a name or properties of the data source. These *stream descriptions* are stored in a built-in stream themselves.

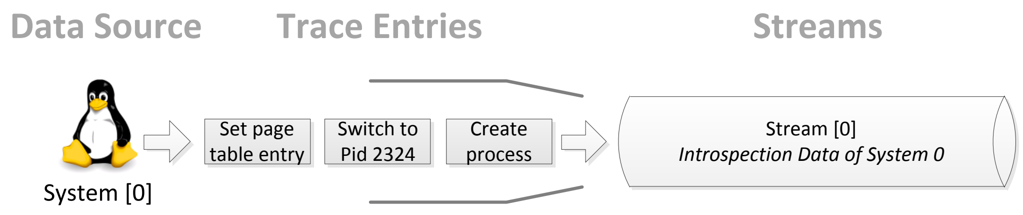


Figure 4.10: Stream Model. Semantically connected trace entries are stored in dedicated streams.

Trace-Lists In contrast to entries and streams, which semantically structure the trace data, *trace-lists* are exclusively used to optimize data storage and access. Their purpose is to partition each stream into a set of segments with a pre-configured amount of entries (typically approx. 2.5 million $\triangleq$ 24 MiB at 10 byte per entry). These segments can then be processed (i.e., read or written) independently of each other.

As already mentioned, tracing all needed information is a costly operation with regard to I/O bandwidth and storage space. Therefore, the entries in each trace list are compressed using the Lempel–Ziv–Markov chain algorithm (LZMA) from the 7-Zip compression library [2]. Naturally, the achievable space savings heavily depends on the data which is to be compressed. For trace data it varies between 80% and 99%. The latter is realistic if the page frame hashes for each CPU write operation are omitted. They hurt the compression ratio because of the high degree of entropy they induce into the trace data. Compressing the trace lists is the key to make the amount of data manageable. It also takes the burden off the I/O backend as substantially less data needs to be written to disk.

Since the simulation of the target system is single-threaded, a modern quad-core processor provides enough computational power to perform an asynchronous compression of trace lists on the spare cores. The independence of trace lists enables the tracer to compress multiple trace lists simultaneously. On the other side, trace lists allow the analyzer software to only partially decompress a stream (in parallel if multiple trace lists are involved). This is very important as the total size of a trace file quickly exceeds the available memory capacity.

Despite the compression, trace lists also function as index for the analyzer. Since each piece of trace data may span multiple trace entries, the exact position of certain trace data in the stream is unknown. Instead, it is necessary to search the first trace-entry of the record by iterating over all previous trace entries. Thus, the random-access performance within a stream is very poor. To overcome this limitation, each trace list stores the index of the first trace record it holds. This substantially reduces the number of trace entries that need to be scanned, because the search is confined to the trace list which contains the data of interest. To further reduce seek times the analyzer successively builds an index-based jump table for each list when entries are being searched.

Trace File The last storage primitive used during tracing is the output file which contains all trace data of a single simulation. In its current form, the format of the trace file is kept relatively simple. Figure 4.11 illustrates the layout. Each time a trace list of an arbitrary stream is full, i.e., stores enough trace entries, the trace list is asynchronously compressed and appended to the file. Each list is preceded by a small header that stores various list properties such as the size of the compressed and uncompressed data as well as the index of the first trace-entry contained in the list. To ensure a correct order when the trace lists are read by the analyzer,

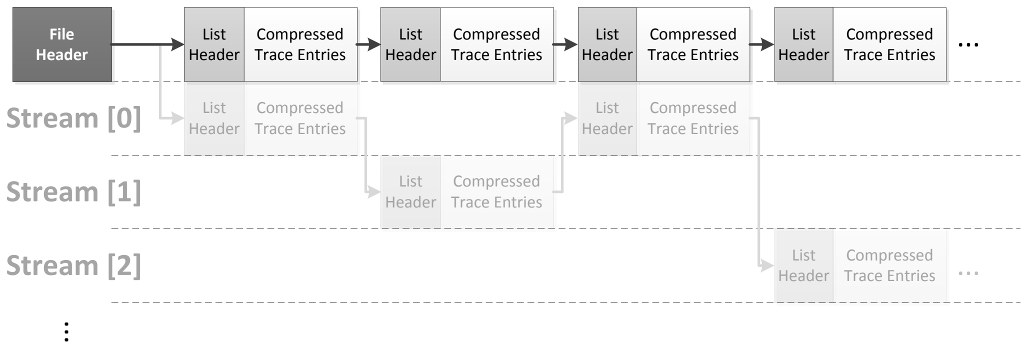


Figure 4.11: Trace File Format. The trace is stored as a linked list of variable-sized trace-lists. Each list comprises a header and compressed trace-entries. The lists are tagged to associate them with the corresponding streams.

a sequence number is included in each header. In addition, an identifier assigns each trace-list to its stream.

4.2.5 Conclusion

The first step when analyzing sharing opportunities is to gather information about which page frames hold equal contents and what temporal and spatial properties these groups of frames have. The tracer accomplishes this through the inspection and continuous monitoring of write accesses to the simulation's physical memory and introduces *sharing groups* to describe and track sets of identical page frames. A red-black tree and hardware accelerated CRC-32C hashing is used to optimize this operation.

However, the information on sharing groups does not enable the analyzer software to examine if for instance certain processes tend to show more sharing opportunities than others of if memory duplication can particularly be observed in address space areas of a certain type. A semantic gap between the simulation and the tracer prevents the retrieval of such information. *Operating system introspection* is able to close this gap by communicating relevant events (e.g., the creation of a new process or address space area) and operating system state information to the tracer. A lightweight communication channel on the basis of a *magic instruction* is employed to transport the data.

To measure the effects of memory deduplication on the performance of processor caches, Simics offers the *g-cache* class which allows simulating arbitrary cache hierarchies. The g-cache has been extended to provide memory duplication related statistics such as the number of cache lines that reference mergeable page frames. Besides basic cache performance metrics such as the current hit-rate, the tracer periodically captures this additional information. Since the simulation is deterministic, comparing the measurements of two subsequent simulations (one with and one without activated memory deduplication) allows examining the effects on processor caches.

Tracing all the mentioned information quickly leads to trace files that are several TiB in size and that get overly complicated to handle. For a single simulated minute of a single-core 20 MHz processor a billion trace-entries (10 bytes each) are generated. The implemented storage system overcomes this challenge through a trace data compression based on LZMA [2] that saves up to 99% of storage space. Furthermore, the applied storage format efficiently organizes semantically connected entries into *streams* and allows the partial decompression of trace data through *trace-lists*.

4.3 Data Analysis

The software design strictly separates the data acquisition from the data analysis phase. This way, time consuming simulations of a target system need only be run once, even if the collected data should be analyzed in many different ways.

In contrast to the tracer, which is implemented as a C-based Simics extension, the analyzer software is completely self-contained and detached from Simics. Thus, it does not need to be adjusted, if the tracer is ported to other simulation platforms. The implementation is based on the Microsoft .Net Framework 4.0 [36] and has been entirely developed in C#. To give access to the core functionality from within other software (e.g., a command line utility), the trace file handling, data interface and full analysis support is encapsulated in a dedicated .Net class library. For the rest of this work this library is referred to as the *core analysis library*.

The fundamental design behind this core library is to provide all means to analyze trace file data, but not to supply any functionality specific to a certain analysis. Instead, a set of generic data primitives and mechanisms is used to supply an interface that works similar to the query-based interface employed by many databases (e.g., SQL). However, the interface provided by the core library is more powerful in that, through the integration of a C#-based scripting environment, it allows to implement rich analysis logic that goes beyond that what usual query-based interfaces offer. In addition, the core library comprises several state machines that are able to reconstruct the operating system state (as described in Section 4.2.2) of the traced machines for any chosen point of simulation time, thereby facilitating a full replay of the traced operating system operations.

This Section (4.3) deals exclusively with the architecture and facilities of the core library. The *trace viewer* tool which adds a graphical user interface to the library's functionality is presented in Section 4.4.

4.3.1 Data Primitives

To explain how trace file data can be analyzed with our library, the data primitives on which the core functions are based on, need to be covered first. These data primitives are very similar to the ones used by the tracer. However, the application of each primitive is extended to increase its flexibility with regard to the analysis' work flow:

Entries An *entry* is the smallest unit of data. All information that needs to be processed by the core functions must be represented as an entry of a specific type. In contrast to the tracer's view of an entry, the analyzer always regards a complete trace datum as a single entry. Thus, if the tracer needed to encode a large datum with the help of multiple 10 byte trace-entries (e.g., a long string or array), the analyzer detects this and outputs only a single entry that rebuilds the entire original information. For that reason, the core library includes over 50 different types of basic entries to cover the spectrum of information captured by the tracer (e.g., data reads/writes, various operating system events, value types, etc.).

In addition to the direct representation of traced data, entries may also represent complex objects such as the model of an entire address space or the reconstruction of a process that was simulated. The approach is to extract as much information as possible from a collection of basic entries and to build semantically richer ones through merging or interpretation. The current implementation comes with approximately 30 of such *extended entries*. Among these are for instance entries to represent sharing groups, processes, address spaces and address space areas.

On the other end of the spectrum, the analyzer is also able to represent basic data types such as an integer or float as an entry.

Streams While in the tracer, *streams* are used as storage destination, the analyzer treats them as a *source* of entries. Streams enable the core library to expose the same interface and fundamental behavior to processing functions regardless of the type of data source that generates the entries. To give special consideration to the different characteristics of data sources, streams incorporate several capability flags that specify how a stream delivers entries (e.g., ordered) and how it may be accessed (e.g., multi-threaded). The latter also includes if a stream can be accessed in random order (such as a list or array) or if a reader is limited to strict FIFO access.

A standard stream expects its data source to already have the entries generated when the stream is read. Therefore, the access to a standard stream is always non-blocking. However, there are cases where a data source is not able to create all entries beforehand. An example is a stream that delivers an entry for each process creation during a running replay of a traced operating system state. To enable analysis code to inspect the current state, whenever a new process is created, it must be able to start reading the stream at the beginning of the replay. In that case, reading the stream must be a blocking operation so the analysis code is paused until the replay reaches the creation of a new process and a respective entry is created. Figure 4.12 depicts the logic of a *blocking stream*:

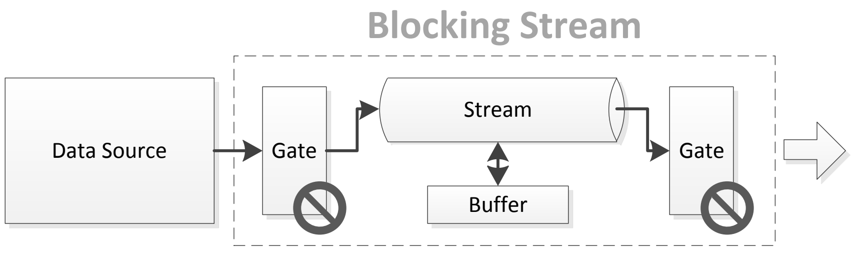


Figure 4.12: Blocking Stream. A blocking stream works like a pipe in UNIX. An intermediary buffer stores entries. If the buffer is full, the data source is blocked. If it is empty, the reader is blocked.

The data source and the reader can block on a blocking stream whenever the intermediary buffer, which holds the entries, is full or empty (depending on the operation). Blocking streams, thus, very much resemble UNIX pipes. Consequently, reading a blocking stream removes the entries from the stream, whereas this is not the case for standard streams.

Another special type of stream is the *multiplexing stream* that merges multiple input streams into a single output stream. The input streams need to be either all non-blocking, in which case the entries are returned according to their timestamp, or blocking where entries are served in the order they reach the multiplexing logic.

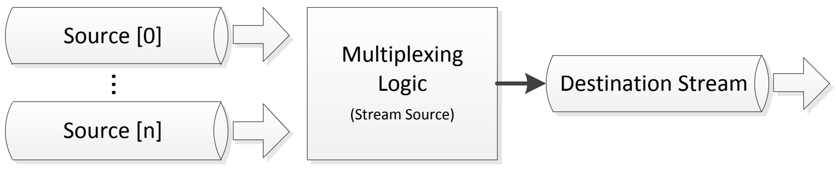


Figure 4.13: Stream Multiplexing. Multiple source streams are combined into a single output stream.

Stream Sources Streams are typically provided by objects that identify themselves as a *stream source*. A trace file for instance is just a stream source that gives access to the streams which the tracer generated and which map to entries stored in the respective file on disk. An example for another stream source is an address space that offers access to its virtual pages and mapped address space areas through streams. Since

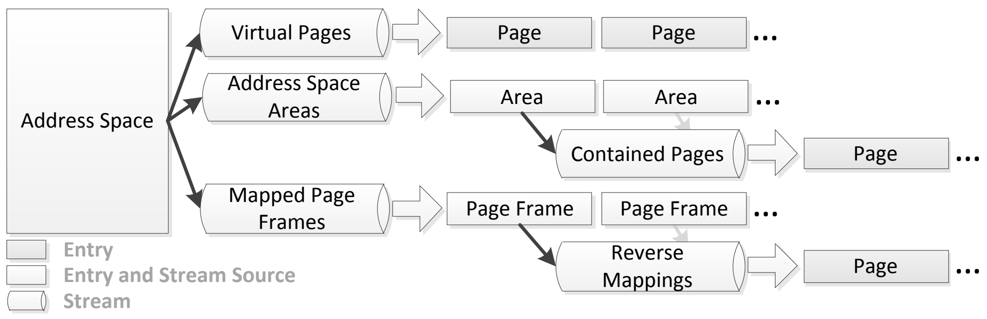


Figure 4.14: Stream Sources. Streams are created by stream sources. Each entry within a stream can be a stream source itself and offer further streams.

each stream source is automatically an entry, streams may deliver entries that are stream sources themselves and which expose further streams. A mapped address space area for instance is not just an entry, but also a stream source that via a stream gives access to particularly those pages of the parent address space that it contains.

Trace The counterpart to a trace file in the tracer is the *trace* object. It functions as working context for a single trace file and brings together all the objects and components that are created as part of the file's analysis. Among these components are various caches to improve analysis performance, the host for the scripting environment and the stream management.

4.3.2 Queries

Up to this point, the presented primitives do not provide any means to analyze the data they hold. *Queries* close this gap. They constitute the mechanism around which any analysis is centered and which enables to do arbitrary computations on trace data.

Figure 4.15 depicts the basic concept. A query is always directed to a single input stream. When the query executes, the entries of this stream are sent through multiple stages of a user-defined filtering and analysis logic. Entries that are returned by the logic are again accessible as a stream and can be used as input for further queries.

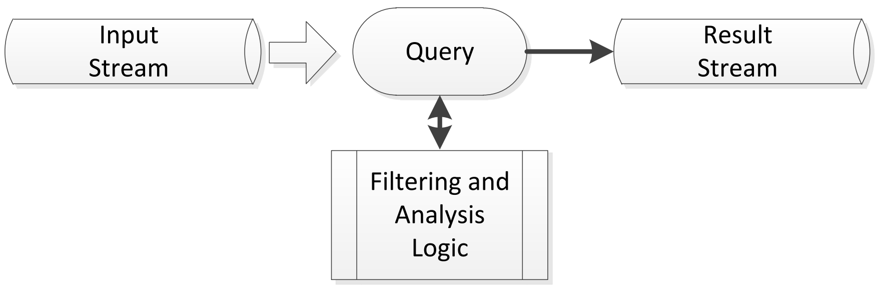


Figure 4.15: Analysis through Queries. Entries from an input stream are processed through custom query logic.

The result entries returned by the filtering and analysis logic do not need to be input entries, but instead can be completely new ones from one or more entirely different types. The amount of entries is also independent from that of the input stream. This way, any data present in a (non-blocking) stream can be analyzed in different ways and iteratively be transformed into entries with richer informational value with regard to the desirable analysis result. To compute a histogram of the lifetime distribution of sharing groups, for instance, a query can be executed that takes a stream of sharing groups as input, computes the histogram in its analysis logic and returns entries representing the individual values of the histogram in its result stream. This stream can then be used as input for further queries or it can be exported to process it with external software such as a plotter. In the following sections this example is used to demonstrate the structure of a query.

Depending on the capabilities of the input stream, queries can be configured to process only a selected window of the stream (based on index or timestamp). Moreover, the maximum number of entries returned can be limited. This is useful if the output should be confined in size but the amount of entries in the window of interest is not known beforehand.

Filtering and Analysis Logic

The processing of entries through a query is divided into two distinct stages. Custom filtering and analysis logic can be applied at both or only one of the stages[2]:

First-Stage Processing The first-stage processing logic is executed in a dedicated run for each input entry. The first stage, thus, works on single input entries, only. Independent from the computations performed by the logic, for each entry the logic ultimately reverts to one of the following three basic operations:

- Pass the entry on to the next stage
- Create an entirely new entry (of any type) and pass that on
- Block the entry so it is not processed any further

The execution of the filtering and analysis logic is not stateless. Thus, the code may generate any kind of state information and access it in subsequent executions. However, the state is only accessible within the current

[2] In fact, it is possible to not set any analysis logic at all. In that case the selected input window (if any specific) is returned one-to-one.

query and solely for the time of the query's execution. This includes, sharing the state with the second-stage logic.

To compute the histogram of the lifetime distribution of sharing groups, the first-stage analysis code would start by initializing an empty histogram as part of its state. The query would then invoke the analysis logic for every sharing group in the input stream, thereby enabling it to inspect each sharing group's lifetime and to update the histogram accordingly. Since the entries themselves are not required in the second stage, the logic would block all entries, effectively returning no data. Instead, the associated state would hold all information, ready to be further processed by the second stage.

To optimize performance, the first-stage processing is parallelized[3]. If, for instance, the retrieval of entries from the input stream involves much computational overhead this can substantially reduce total execution time. The streams directly provided by a trace file are a good example, as, on access, they trigger the decompression of relevant trace-lists. The parallelization spreads this operation over all available CPU cores by finding windows in the input stream that can be processed concurrently. Therefore, any analysis code for the first-stage processing has to be developed with thread-safety in mind. Moreover, the logic must not depend on any order in which entries are supplied.

To sum up, the first stage should be used to discard entries as quickly as possible (build richer state information instead) and to do operations that benefit from parallelization.

Second-Stage Processing The second-stage analysis logic is executed when all entries have been processed by the first stage and the results have been trimmed to the specified amount of maximum allowed query results (unlimited by default). The second stage is very similar to the first one in its ability to freely operate on the input entries. This again includes creating completely new entries. In contrast to the first stage, however, the entries are now guaranteed to be ordered by their index/timestamp and are delivered in custom-sized groups. By default all entries are served in a single large group, thereby facilitating computations with access to all entries at once. This is useful to implement analysis logic that is not par-

[3]Parallelization is suppressed if the input stream does not support multi-threaded, indexed access. This is for example the case for blocking streams.

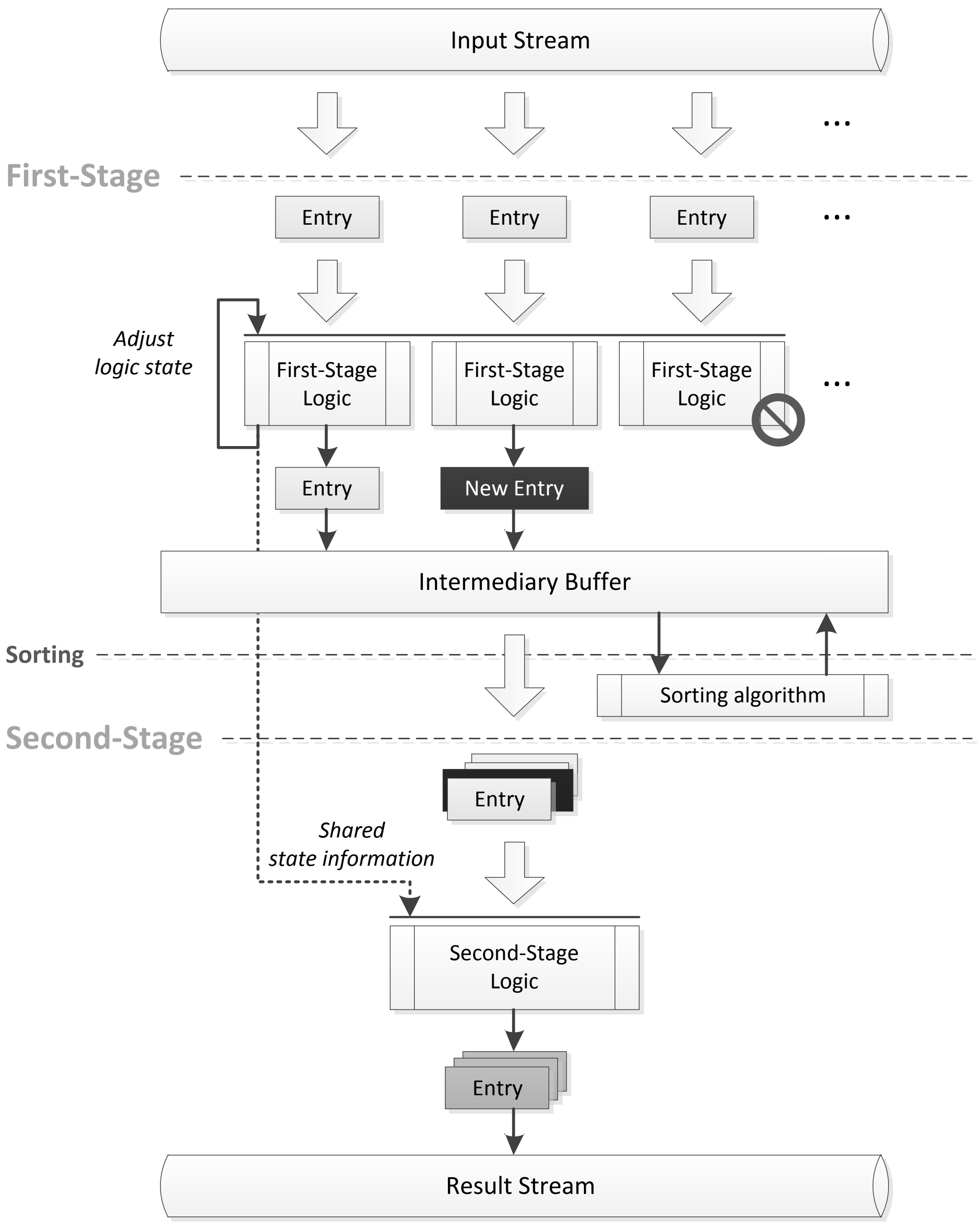


Figure 4.16: Two-Stage Query Processing. The first stage logic is executed in parallel on a single entry each. The first stage can pass an entry on, create a new one or block it. Entries are gathered in an intermediary buffer and are sorted if necessary after the first stage has finished. They are then sent (in custom-sized groups) to stage two where arbitrary operations on the ordered entries may be done. The output is stored in a new stream.

allelizable while still allowing the multi-threaded retrieval of entries in the first stage. If a group size is specified, the computations are parallelized on a per-group granularity.

To finish the computation of the histogram showing the lifetime distribution of sharing groups, the second-stage logic would take the histogram from the shared state and create an (integer) entry for each data point.

Implementation of Custom Analysis Logic

Similar to a query in a SQL database, queries allow filtering, combining and inspecting entries. However, the filtering and analysis logic of a query is not restricted to a single statement. Instead, it is implemented with the help of a C#-based object-oriented scripting environment, which is integrated into the core library via CS-Script [46]. The environment exposes full access to the functionality of the core library as well as to the entire .Net Framework, including Language Integrated Query (LINQ) [32] and 3rd-party .Net assemblies. To preserve performance, all C# scripts are compiled prior to their execution and therefore run as fast as code directly built into the core library.

An alternative to using C# scripts is to implement the logic in a dedicated .Net assembly. However, this requires restarting the entire analyzer if changes to the code need to be performed. Scripts, in contrast, can be modified on-the-fly and thereby make it substantially easier to experiment with variations of the analysis logic (e.g., tuning a parameter).

Listing 4.1 illustrates the structure of an analysis script. The logic is encapsulated in a dedicated class, in the listing called `MyAnalysisLogic`. The class implements first-stage and second-stage analysis through the public methods `FSPFilterMain` (**F**irst - **S**tage **P**rocessing) and `SSPFilterMain` (**S**econd - **S**tage **P**rocessing), respectively. In the listing both methods just pass the input on. The first-stage method can replace the input entry by overwriting the `entry` argument. The return value determines blocking. The second-stage method controls its output solely through the returned list of entries. The `parameter` argument supplies an optionally configured user-value. In contrast to the listing, the methods are not required to reside in the same script. For every query, the core library creates a new instance of the class(es) that implement the analysis logic. It is therefore possible to use class members to store and share state information and to use the class constructor to perform state initialization.

```
using System;
using System.Linq;
using System.Collections.Generic;
using System.Text;

using TraceViewer.Core;
using TraceViewer.Core.Entries;
using TraceViewer.Core.Entries.Linux;
using TraceViewer.Core.Filtering;
using TraceViewer.Core.Scripting;
using TraceViewer.Core.Statistics;

public class MyAnalysisLogic :
  ICustomFSPFilter,
  ICustomSSPFilter
{
  private int _myStateVariable;

  public MyAnalysisLogic()
  {
    // Initialize state here

    _myStateVariable = 1;
  }

  public bool FSPFilterMain(ref Entry entry,
    object parameter)
  {
    // Implement first-stage processing here

    return true; // Return false to block the entry
  }

  public List<Entry> SSPFilterMain(
    List<Entry> entries, object parameter)
  {
    // Implement second-stage processing here

    return entries;
  }
}
```

Listing 4.1: Template Analysis Script

4.3.3 Data Correlation

Queries provide a powerful mechanism to do filtering and arbitrary computations on entries. However, queries are conceptually very generic and thereby lack a convenient way to directly work on the raw, basic trace data gathered during the simulation. It is, for instance, not possible to easily answer the question what processes were involved in certain sharing opportunities. Therefore, queries need to be supplemented with a mechanism that is specifically directed to work on the collected raw trace data and that is able to build richer information from it.

The approach taken in this work is to use the trace data from memory inspection and operating system introspection, reconstruct relevant parts of the simulated machine's state and present this richer information in a form suitable to be analyzed with queries. This way, analysis logic can, for instance, easily get the processes that were alive at a certain point in time within the simulation and correlate these with the sharing groups that were concurrently present.

State Models

To reconstruct relevant parts of the simulated machine's state, including simulated virtual machines, a model is needed that is capable to reproduce the state information of interest. Depending on the state's scope this model can reach a substantial complexity. The core library therefore reduces the complexity by splitting the state model in multiple small models that each cover independent areas of the overall model. This approach also comes with the benefit of a natural modularization that increases flexibility and extensibility. The current implementation includes three state models:

System State Model The system state model interprets trace data that is generated by operating system introspection (see Section 4.2.2). For every traced point in time it is able to reproduce major state information of the operating system that was running in the simulated host (or virtual machine). The following list gives an overview of the most important information available:

- A hierarchy of all processes and kernel threads alive (incl. name, command line, environment variables, etc.)

- The complete address space layout for each process (heap, stack, mapped files, etc.) and for the kernel (kernel memory pools, kernel stacks, file cache, etc.)
- The mapping of virtual pages to physical pages for each address space including complete reverse mapping information
- Page frame allocation (free/used status for each frame)
- The currently running process and active address space

In addition, the system state model is capable to traverse reverse mapping information to infer per-page frame usage information (e.g., frame is used as heap).

As already depicted in Figure 4.14, most of the system abstractions implemented by the system state model such as processes, address spaces and individual virtual as well as physical pages are entries that are available in streams and, thus, can be analyzed via queries. Furthermore, they are directly accessible through public members so that analysis code always has access to the whole model.

To keep the number of abstractions small, the physical memory is represented through an address space entry just like a standard process address space. The same is true for the kernel address space. However, the latter is additionally set as overlay for the kernel address range in process address spaces so that operations (e.g., a write or mapping) targeted at kernel memory locations are always redirected to the same kernel address space.

Sharing Model The sharing model is responsible to process entries generated during memory inspection (see Section 4.2.1). Its primary tasks are to reproduce the sharing groups that existed at the targeted point in simulation time and to update page access statistics and content hashes. The following list summarizes the most important information provided by the sharing model:

- The sharing groups (incl. list of referenced frames, pattern (if any) or complete page contents if traced, creation time, stable time[4], etc.)

[4]An object's stable time denotes the time span in simulation time between its last state change and the currently inspected point in simulation time.

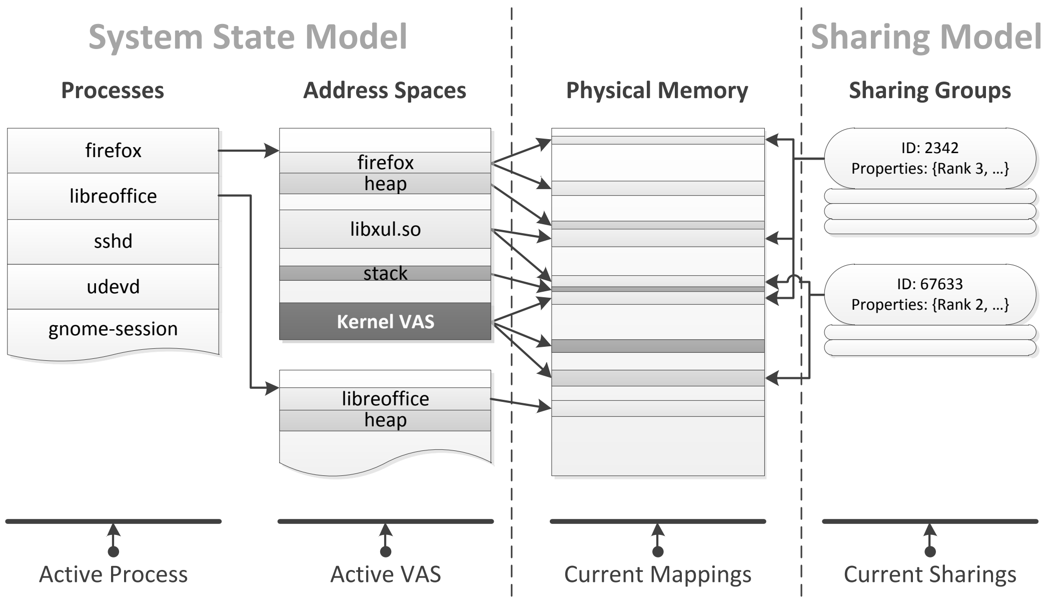


Figure 4.17: System State Model and Sharing Model. The system state model and the sharing model work on the same physical memory address space, thereby connecting information about sharing groups and operating system state.

- A hash of each page frame's current contents
- Per-page frame write statistics (number of writes, percentage of non-destructive writes[5], last modified time, stable time)

Like the system state model, the sharing model works with a physical memory address space. If the system state is evaluated together with the sharing model, both models may use the same physical memory object and thereby connect information about sharing groups and the operating system state. This is illustrated in Figure 4.17. Since the sharing groups are accessible as stream, a query can now be executed which analyzes how sharing groups are related to processes and certain types of address space areas.

Cache Model For some analyses of the cache performance a detailed insight into the assignment of cache lines can be beneficial. This is for example the case if an examination intends to identify processes that utilize

[5]A non-destructive write is a write operation that does not change the contents of a page.

a high number of cache lines to cache identical page frames. The cache model allows to perform such evaluations by reproducing the cache line assignment. This includes a page frame number and offset for each line.

Since the current tracer does not implement a continuous recording of cache internal operations such as line fetches, the cache model is only updated periodically with the interval which has been chosen for tracing.

Resolving

State models only provide abstractions and the operations that can be performed on these abstractions. This is for example the mapping of a virtual page to a physical page in an address space. The mechanism that reads corresponding input streams and ensures that each model in the analysis represents its state at the same point in simulation time is the *resolver*. It is responsible to replay the traced actions in the exact same order they occurred in the simulation, synchronized over all models. The resolver thereby allows jumping to any point within the traced simulation and the attached models are brought into their respective state. This state can then be analyzed with queries that work on the numerous streams offered by the state models.

Figure 4.18 illustrates the process of state resolving. Each state model has a single input stream attached which exclusively contains entries that it is able to interpret. This is, for instance, a stream of introspection data entries for a system state model. To include a specific (stream, state) tuple, it is added to the resolver. The resolver's task is to sort the entries according to the time they were traced in the simulation and to apply them to their corresponding state models afterwards. This process is continued until the specific target simulation time is reached. To sort the entries, the resolver repeats a loop of getting one entry for each stream and forwarding the one with the smallest timestamp. The entries are, thus, required to be sorted within their input streams. Otherwise, the resolver would need to retrieve and sort all entries at once which is not feasible due to the amount of main memory required for such an operation.

Although the presented approach is well suited to analyze sharing opportunities that existed at a certain point in simulation time, it does not provide a convenient way to do analyses that take the whole simulation time into consideration. This might be, for example, an analysis of the average lifetime of sharing opportunities. Therefore, a mechanism is needed that allows

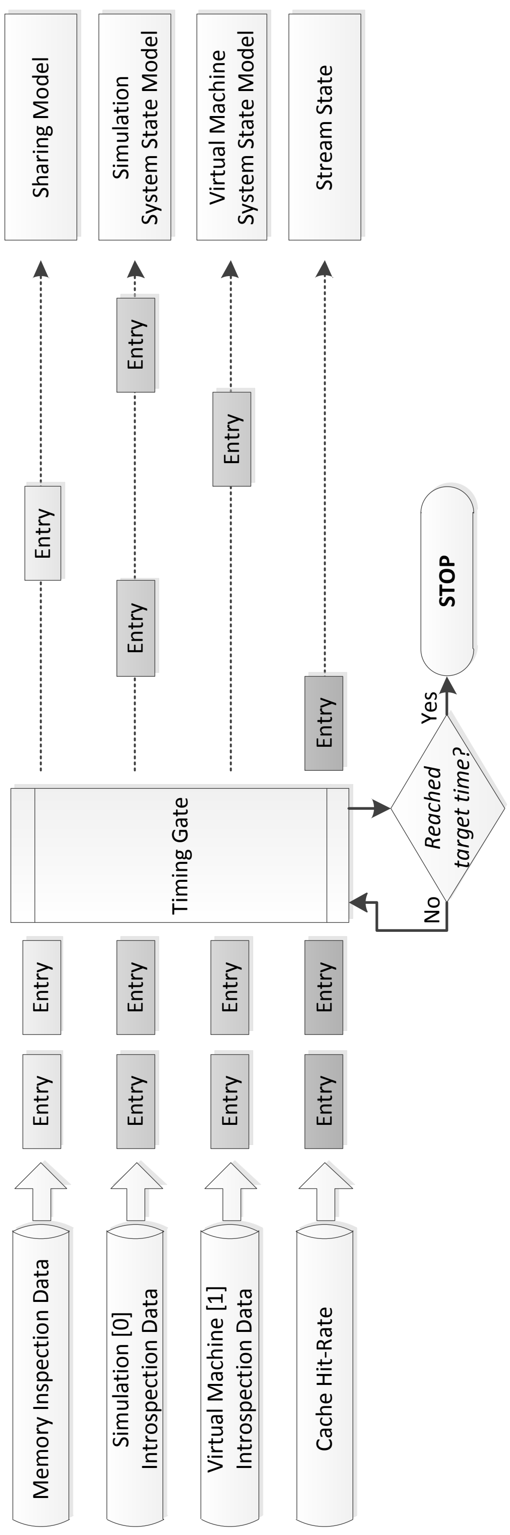

Memory Inspection Data
Simulation [0] Introspection Data
Virtual Machine [1] Introspection Data
Cache Hit-Rate
Entry
Timing Gate
Reached target time?
No
Yes
STOP
Sharing Model
Simulation System State Model
Virtual Machine System State Model
Stream State

Figure 4.18: Resolving. Each state model is attached to an input stream. A timing gate sorts the input entries according to their timestamp. The entries are then applied to their respective state model. Stream states allow the inclusion of arbitrary streams.

analysis code to be invoked at interesting points *during the resolving*. This way, the resolver can be instructed to fully replay all traced actions (i.e., jump to the end of the traced simulation time) and the analysis code is able to concurrently examine the development of states and their abstractions.

The core library offers two mechanisms to perform full simulation time spanning analyses. Since analyses are done with the help of queries, two types of blocking streams are available that are supplies with entries in the course of the simulation replay:

Event Streams For most analyses it is the best to be invoked at certain events in the replay that the researcher intends to examine. The presented state models therefore offer special (blocking) *event streams* on which a query can be executed concurrently to the replay. Each time a state model performs an operation that triggers an event an entry with event-specific information is generated and written to the stream, thereby leading to the invocation of the analysis code. The current implementation only includes events in the sharing model (e.g., creation/destruction of a sharing group).

Although some events are already represented by corresponding entries in the raw traces, entries supplied by event streams typically include information that is not explicitly traced or which would require the cumbersome manual interpretation of a series of basic trace entries.

Stream State Model *Stream states* allow any ordered and time stamped stream to be included in the resolving. This is depicted in Figure 4.18 with the *Cache Hit-Rate* stream. In contrast to the other state models, a stream state model is a pseudo model which only forwards the entries it receives to a blocking stream.

The analysis logic of a query that is executed on one of these types of streams is invoked synchronously to the entry flow during resolving. This way, the logic is able to inspect the entry that triggered it together with the reproduced simulation state at the respective point in simulation time. To compute, for instance, the average lifetime of sharing opportunities, a query can be started on a stream that is supplied with a certain event entry every time a sharing group is destroyed in the sharing model. The analysis code can then compute the lifetime of the dead sharing group and include the new value in its calculations. Another example is the triggering of the analysis for each entry in a stream included via a stream state (e.g., for each measuring of the cache hit-rate). When the analysis has finished processing the entry, the resolver continues the simulation replay.

The resolving of a whole simulation can take considerable time. A sharing model, for instance, reproduces every single CPU write to update statistics and page hashes. On a 3.4 GHz Core-i7 it is realistic that a single simulation replay of 30 minutes (single trace) takes about 3 to 4 hours. Although this is substantially faster than running the simulation itself several times (approx. 1.5 days for a single run), it is beneficial to execute multiple queries concurrently if different analyses are planned. This is no problem as it has been taken into consideration during the design of the relevant mechanisms. It is even possible to execute multiple queries on the same event stream.

Multiple-Trace-Support

Since the Simics version used in this work does not ship processor models that provide the virtualization instruction set extensions Intel-VT or AMD-V, QEMU has been used to implement virtual machine support through binary translation (see Section 4.2.2). However, performing binary translation in the simulation turned out to be a major performance bottleneck. The computational overhead resulted in an additional slowdown factor of up to 300 between the simulated virtualization host and the virtual machine. At the aforementioned simulation time of about 1.5 days for 30 minutes, this made the evaluation of a virtualization scenario unfeasible. It is therefore crucial to use a full system simulator that is capable to emulate virtualization technologies so computational overhead for the virtual machine monitor is kept minimal. This would also benefit the tracing performance, as the additional hop through the VMM could be saved.

An alternative to use virtualization within the simulation is to perform multiple dedicated simulations and interpret these simulations as virtual machines in the analysis. Since the sharing model already keeps page frame hashes up to date, it is feasible to extend the model to find identical page frames between multiple physical memory address spaces through a comparison of the page frame hashes. The core library, thus, has been extended to support the analysis of multiple traces at the same time. This includes the ability to mix streams and state models of different traces in the same resolving process, effectively allowing the concurrent replay of

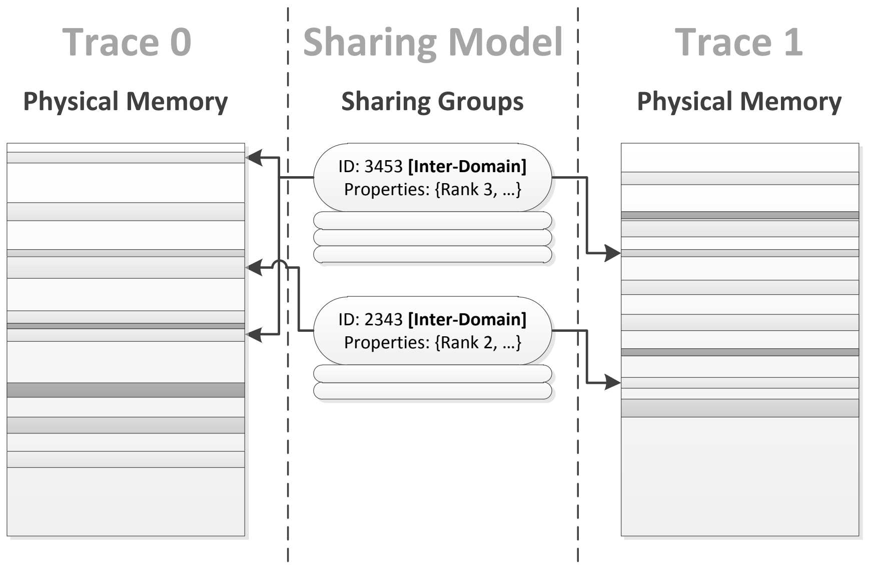


Figure 4.19: Inter-Domain Sharing Groups. If sharing models for different traces are used in the same resolving, inter-domain sharing groups are detected based on page frame hashes.

multiple simulations. If a sharing model detects that other sharing models are involved in the same resolving, the detection of identical page frames between the different physical memory address spaces is activated. Every time a page frame is changed (i.e., its hash is updated), the sharings are updated accordingly. To represent this new form of sharing opportunities, the core library differentiates regular *intra-domain sharing groups* that exist only within a single trace and *inter-domain sharing groups* which cross multiple traces.

Since both types of sharing groups derive from the same base class, analysis code does not need to be adjusted to take inter-domain sharing groups into account. However, the query that does the analysis needs to be executed on a stream that multiplexes the event streams of all sharing models. Otherwise, events that are triggered through additional sharing models are missed.

A drawback of this approach is the chance of false positives during the detection of inter-domain sharing groups due to hash collisions. Using a stronger hashing algorithm than CRC-32C, however, can reduce the risk to a negligible level.

4.3.4 Conclusion

The data analysis code is implemented in a dedicated *analysis core library* written in C# for the Microsoft .Net Framework 4.0. The core library does not supply the functionality specific to a certain analysis but instead provides generic data primitives and mechanisms that are directed to do arbitrary analyses on trace file data.

The data primitives employed in the core library are comparable to that of the tracer, but are extended in their applicability. *Entries* are not restricted to only represent the data included in a trace file but also complex objects such as the model of an entire address space or the reconstruction of a process that was simulated. *Streams* are utilized as universal data store and transport channel for entries of any type. In contrast to the tracer, any object can identify itself as *stream source*, and thereby publish own streams (e.g., a stream of pages in an address space).

Queries are a generic way to analyze the entries in a stream with the help of C# object-oriented scripts that have access to the full .Net Framework and 3rd-party assemblies. They enable to do powerful filtering and arbitrary computations on entries. Query results are accessible again as a stream, thereby enabling information enrichment through iterative processing.

To interpret and connect trace data recorded through memory inspection and operating system introspection, a simulation replay mechanism is available. It employs various *state models* to reproduce relevant parts of the simulated machine's state that existed at a certain point in simulation time. This includes, besides others, the process hierarchy and the layout of the processes' address spaces including virtual-to-physical page mappings as well as the list of sharing groups. Queries are used to inspect the state information and thereby allow making statements about the connection of sharing groups and operating system abstractions such as processes and address space areas. Besides inspecting a certain point in simulation time, *resolving* also enables to do full simulation time spanning analyses (e.g., the computation of the average lifetime of sharing groups). Moreover, it allows performing analyses that include multiple trace files at once.

When combined, the presented mechanisms provide a powerful toolset that enables a researcher to quickly implement custom analyses, covering even a broader field than the analysis of sharing opportunities alone.

4.4 Trace Viewer

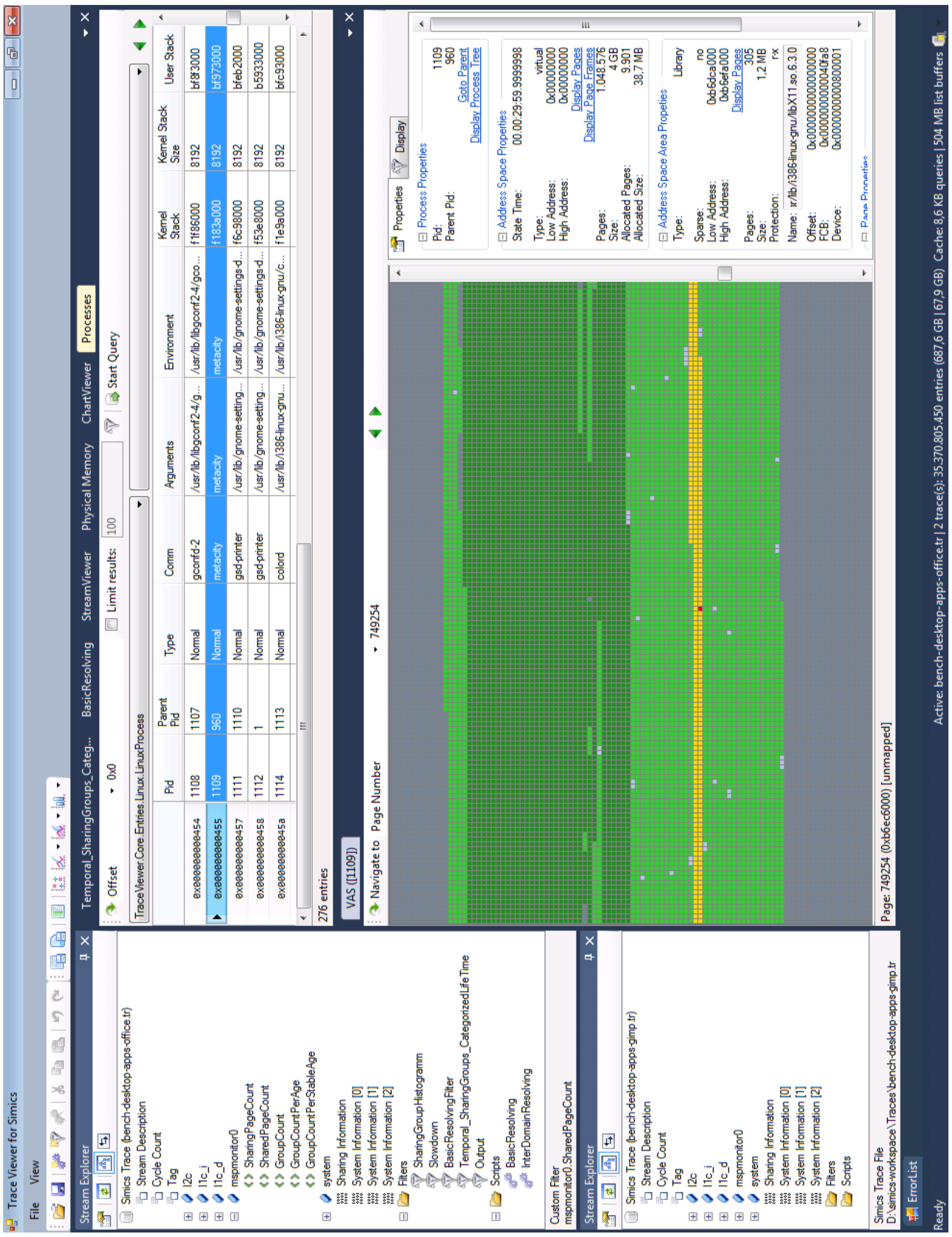
The presented core library implements the mechanisms to open and work with trace files. The *trace viewer* tool supplements the core library by providing a graphical user interface to access the analysis functionality. Figures 4.20 and 4.21 depict the tool's user interface.

The GUI is primarily based on a range of tool windows that allow the user to work with and inspect trace data. Since each tool window works on a single instance of a certain primitive (e.g., stream) or abstraction (e.g., address space), it is possible to have multiple instances of the same type of tool window open concurrently (e.g., to inspect two different streams). A flexible docking and tabbing system allows clearly arranging the windows and making use of the extra space available in multi-monitor setups. Potentially time-consuming operations such as the execution of a query or the export of a stream are performed in the background and therefore allow the user to continue working while waiting for results. In addition, all operations are cancelable at any time.

In its current state the trace viewer implements five major tool windows:

Stream Explorer The core library supports to open more than one trace file at the same time to do analyses that take multiple traces into account. For each open trace file a *stream explorer* window is created. It enumerates the streams contained in a trace file and builds a tree representing the hierarchy of the corresponding data sources traced in the simulation. Besides streams, the explorer also lists analysis scripts (under Filters) and general scripts. Double-clicking any of the elements opens the according viewer or editor.

Stream Viewer To view the entries contained in a stream, the *stream viewer* is the right tool window. It allows performing a query on the supplied stream and subsequently displays the result entries in a table. By default, the query is not configured with any analysis logic and thus simply returns the input entries. However, even with a one-to-one mapping the query can be used to restrict the display to a specific window within the stream. If analysis scripts are configured, the stream viewer is a convenient way to quickly get a view on the analysis results.


Trace Viewer for Simics

File View

Stream Explorer

Simics Trace (bench-desktop-apps-office.tr)
- Stream Description
- Cycle Count
- Tag
- l2c
- l1c_i
- l1c_d
- mspmonitor0
  - SharingPageCount
  - SharedPageCount
  - GroupCount
  - GroupCountPerAge
  - GroupCountPerStableAge
- system
- Sharing Information
- System Information [0]
- System Information [1]
- System Information [2]
- Filters
  - SharingGroupHistogramm
  - Slowdown
  - BasicResolvingFilter
  - Temporal_SharingGroups_CategorizedLifeTime
  - Output
- Scripts
  - BasicResolving
  - InterDomainResolving

Custom Filter
mspmonitor0.SharedPageCount

Stream Explorer

Simics Trace (bench-desktop-apps-gimp.tr)
- Stream Description
- Cycle Count
- Tag
- l2c
- l1c_i
- l1c_d
- mspmonitor0
- system
- Sharing Information
- System Information [0]
- System Information [1]
- System Information [2]
- Filters
- Scripts

Simics Trace File
D:\simics-workspace\Traces\bench-desktop-apps-gimp.tr

Temporal_SharingGroups_Categ... BasicResolving StreamViewer Physical Memory ChartViewer Processes

Offset 0x0 Limit results: 100 Start Query

TraceViewer.Core.Entries.Linux.LinuxProcess

	Pid	Parent Pid	Type	Comm	Arguments	Environment	Kernel Stack	Kernel Stack Size	User Stack
0x000000000454	1108	1107	Normal	gconfd-2	/usr/lib/libgconf2-4/g...	/usr/lib/libgconf2-4/gco...	f1f86000	8192	bf8f3000
0x000000000455	1109	960	Normal	metacity	metacity	metacity	f183a000	8192	bf973000
0x000000000457	1111	1110	Normal	gsd-printer	/usr/lib/gnome-setting...	/usr/lib/gnome-settings-d...	f6c98000	8192	bfeb2000
0x000000000458	1112	1	Normal	gsd-printer	/usr/lib/gnome-setting...	/usr/lib/gnome-settings-d...	f53e8000	8192	b5933000
0x00000000045a	1114	1113	Normal	colord	/usr/lib/i386-linux-gnu...	/usr/lib/i386-linux-gnu/c...	f1e9a000	8192	bfc93000

276 entries

VAS ([1109])

Navigate to Page Number 749254

Properties Display

Process Properties

Pid:	1109
Parent Pid:	960
	Goto Parent
	Display Process Tree

Address Space Properties

State Time:	00.00:29:59.9999998
Type:	virtual
Low Address:	0x00000000
High Address:	0x00000000
	Display Pages
	Display Page Frames
Pages:	1.048.576
Size:	4 GB
Allocated Pages:	9.901
Allocated Size:	38,7 MB

Address Space Area Propeties

Type:	Library
Sparse:	no
Low Address:	0xb6dca000
High Address:	0xb6efa000
	Display Pages
Pages:	305
Size:	1,2 MB
Protection:	r-x
Name:	sr/lib/i386-linux-gnu/libX11.so.6.3.0
Offset:	0x0000000000000000
FCB:	0x00000000000040fa8
Device:	0x0000000000800001

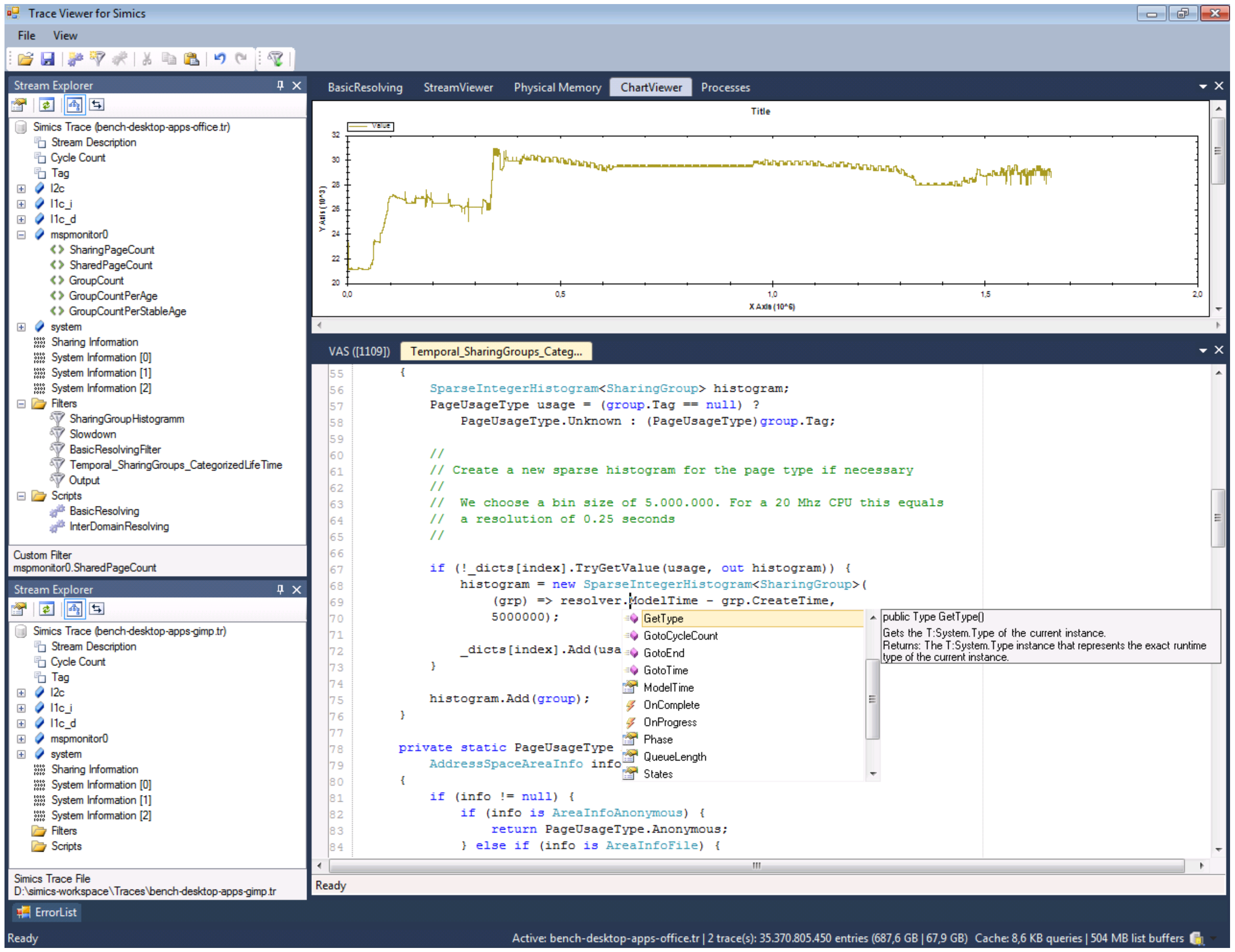
Page Properties

Page: 749254 (0xb6ec6000) [unmapped]

ErrorList

Ready Active: bench-desktop-apps-office.tr | 2 trace(s): 35.370.805.450 entries (687,6 GB | 67,9 GB) Cache: 8,6 KB queries | 504 MB list buffers

Figure 4.20: Trace Viewer. Left: Stream explorers showing streams of two trace files; Top: Stream viewer, listing processes after resolving; Bottom: Address space viewer, showing address space layout of the *metacity* process and details on a selected area and page.



```
    {
        SparseIntegerHistogram<SharingGroup> histogram;
        PageUsageType usage = (group.Tag == null) ?
            PageUsageType.Unknown : (PageUsageType)group.Tag;

        //
        // Create a new sparse histogram for the page type if necessary
        //
        //  We choose a bin size of 5.000.000. For a 20 Mhz CPU this equals
        //  a resolution of 0.25 seconds
        //

        if (!_dicts[index].TryGetValue(usage, out histogram)) {
            histogram = new SparseIntegerHistogram<SharingGroup>(
                (grp) => resolver.ModelTime - grp.CreateTime,
                5000000);

            _dicts[index].Add(usa
        }

        histogram.Add(group);
    }

    private static PageUsageType
        AddressSpaceAreaInfo info
    {
        if (info != null) {
            if (info is AreaInfoAnonymous) {
                return PageUsageType.Anonymous;
            } else if (info is AreaInfoFile) {
```

public Type GetType()
Gets the T:System.Type of the current instance.
Returns: The T:System.Type instance that represents the exact runtime type of the current instance.

Active: bench-desktop-apps-office.tr | 2 trace(s): 35.370.805.450 entries (687,6 GB | 67,9 GB) Cache: 8,6 KB queries | 504 MB list buffers

Figure 4.21: Trace Viewer. Left: Stream explorers showing streams of two trace files; Top: Chart viewer, illustrating development of the amount of mergeable pages; Bottom: Script editor, presents C# script file with syntax highlighting and code completion.

Address Space Viewer The *address space viewer* is the only tool window specifically designed to give an insight into the properties of a certain type of entry. The primary interface of this window is a visualization of the address space and its individual pages. Groups of pages that semantically belong together such as an address space area or a sharing group are highlighted when the user hovers over them. Different types of groups (e.g., a group of pages forming a heap vs. stack) are displayed in specific colors to make it easy to quickly locate and differentiate them.

The visualization can also be switched between different display modes:

- Address Space Layout (Heap, Stacks, Mapped Files, etc.)
- Page Allocation (Free/Used)
- Page Frame Mapping (Mapped to Page Frame/Unmapped)
- Sharing Groups
- Heatmap or Binary Map (Reads/Writes/Non-Destructive Writes)[6]

Chart Viewer The *chart viewer* is a plotting tool which can visualize integer or float based trace data such as the development of shareable pages as depicted in Figure 4.21. The user can choose from the most common chart types (e.g., scatter chart, line chart, area chart, bar chart) and freely zoom and pan within the plotted data. It is possible to export the plot as pixel or vector graphic. However, the tool is not intended to do advanced plotting but only helps to get a quick visualization of traced or generated data.

Script Editor To implement analysis logic, the *script editor* can be used. The user can compile the scripts from within the editor and a list of errors is presented if the compilation fails. The collapsed *error list* window can be seen in Figure 4.21 at the bottom left corner. Scripts that successfully compile are subsequently available to be used in queries. In addition to analysis logic, the script editor supports to write and directly execute generic scripts that are not supposed to be run in the context of a query. This way, scripts can be written that execute queries and start simulation replays. Since the current version of the trace viewer does not include a

[6]The threshold for the binary map as well as the range for the heatmap are configurable via a slider control.

tool window to perform resolving, a generic script is the only way to start a simulation replay from within the software. To ease the development of scripts, the editor includes syntax highlighting and code completion.

4.5 Simulation Control

Like most virtualization software, Simics uses virtual hard disk images (VDIs) to store the contents of emulated disk devices. To run a benchmark within the simulation, an operating system and the benchmark itself need to be installed into the disk image. Since it is very time consuming and inconvenient to perform an installation in a simulated machine, it is a more efficient way to prepare the disk image with much faster standard virtualization software such as VirtualBox [52]. The disk image can afterwards be converted into a Simics compatible format.

Since a fully tracing simulation is too slow to be interactively controlled, the system needs to be configured to execute the benchmark automatically after boot. If only a single benchmark is supposed to be run, this solution is sufficient. However, this is not the case if multiple benchmarks with differing software or input parameters are planned and a serial execution is not desired (e.g., to ensure the exact same start environment). In that case, a dedicated (explicitly configured) disk image is needed for every benchmark configuration. This complicates the overall management and makes changes to the underlying system cumbersome because multiple disk images need to be maintained.

This work takes a different approach that requires only a single disk image but offers the flexibility to choose the benchmark to be executed *in the simulator*. This is accomplished with the help of a tool (*tracerctrl*) that runs within the simulation and that requests a benchmark script directly from the simulator. Figure 4.22 depicts the sequence of actions. Like the operating system introspection (see Section 4.2.2), the tool utilizes the magic instruction to communicate with the simulator and ultimately with the tracer. The tracer in turn is able to interpret the request, reads a previously for this simulation run configured shell script and writes it into a buffer within the simulation. The buffer has been previously allocated by the tool (in its virtual address space) and its address is supplied as part of the request. When the tool continues execution it can simply write the script into a file and invoke the appropriate shell to run it (e.g., bash).

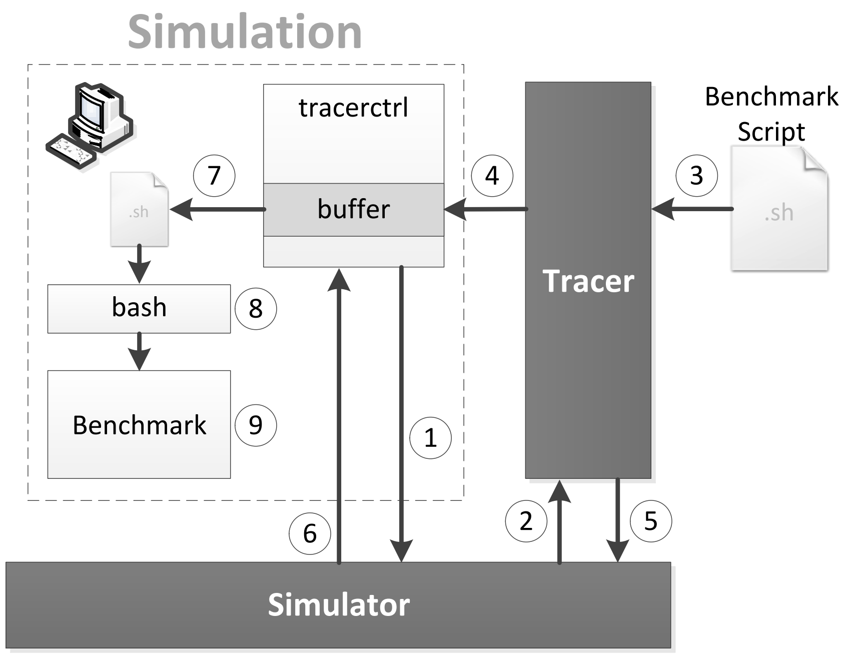


Figure 4.22: Simulation Control. Benchmarks scripts are transferred into the simulation via "shared memory", thereby avoiding changes to the virtual disk image.

For this approach to work, the disk image needs to be prepared to run every application required for the entire set of planned benchmarks. Afterwards, very high workload flexibility is reached, as the benchmark script can be efficiently adjusted and extended without the need to change the virtual disk image.

In its current implementation, the tool also enables to instruct the simulator to execute scripts. This is very useful to, for instance, start full tracing only on certain events within the simulation. Another use case is to stop the simulation when a benchmark run finished. In contrast to the previously mentioned direction, scripts executed by the simulator are required to be already stored on the host. A transfer out of the simulation is not intended.

Another useful feature exposed by the tool is to tag the trace file. A tag is a special trace-entry that facilitates the structuring of a trace from the perspective of the researcher by marking a certain point in simulation time with a custom string message. This way, a benchmark script can, for instance, tag the start and end times of a certain workload. In the current implementation, tags are also used to store the console output of benchmarks in the trace file. Depending on the scenario, this enables to lookup results such as execution times or other performance metrics measured during the benchmark.

4.6 Conclusion

Full system simulators only provide the platform to perform simulations of a target system. To analyze sharing opportunities within the simulation, additional software components are necessary. The software architecture chosen in this work decouples the data acquisition from the data analysis phase. This way, a high flexibility in the analysis is achieved as the time consuming rerun of simulations is avoided if different analyses are planned or if frequent changes to analysis parameters are required. To bridge the gap between the two phases, tracing is employed.

The tracer resides as an extension in the simulator and monitors the simulation's memory, detects any changes to the contents of page frames and identifies and tracks sharing opportunities. The gathered information allows an analysis of the temporal and spatial properties of sharing opportunities. However, they do not enable a correlation of sharing opportunities with the operating system state of the target system such as the usage of page frames. A semantic gap prevents the tracer from inferring such information. To retrieve the desired data the tracer employs operating system introspection. Kernel modifications in the target actively communicate interesting events such as the creation of processes or the allocation of memory areas to the tracer. To transport the data, a lightweight mechanism is used that minimizes the overhead within the simulation by exploiting the ability of full system simulators to instrument CPU instructions. In addition, the tracer makes use of cache simulation to periodically record cache related statistics.

All gathered data is written to a specially formatted and compressed trace file and is thereby made available for a later analysis. Instead of implementing specific analyses, the software components for the examination of trace data provide a set of generic data primitives and mechanisms that give the researcher room to analyze the input data in many different ways. Queries are the central mechanism offered by the analyzer software. They allow filtering trace data and doing arbitrary computations. To implement analysis logic that is executed in the course of queries, specifically crafted

C# scripts are used. However, with queries alone it is difficult to interpret the immense flow of trace data that is collected during the simulation. This makes it hard to examine, for instance, the connection of sharing opportunities with certain processes of address space areas. Thus, the analyzer implements a resolving mechanism that models relevant parts of the traced simulation's operating system and memory state (e.g., address space layouts, virtual-to-physical page mapping, etc.) and is capable to do a replay of these parts of the simulation. With resolving in place, queries can be performed on data that has a much richer informational value and therefore facilitate complex conclusions. A powerful GUI supplements the core analysis functionality and enables to visualize generated examination results.

Chapter 5

Evaluation

In the previous chapters a full system simulation based tracing and analysis software has been proposed as a platform for the fine granular analysis of sharing opportunities. This chapter supplements the presented approach with a prototypical evaluation, which demonstrates the practical feasibility of analyzing sharing opportunities with the proposed design. Moreover, it gives examples on how the functionality of the analysis core library can be used to examine the characteristics of sharing opportunities as described in Chapter 3.

In particular, the evaluation shows that the approach is capable to:

- Examine the temporal and spatial characteristics of sharing opportunities in a resolution, which goes beyond that of previous research.
- Fully correlate the sharing opportunities with the operating system state and allow a detailed semantic differentiation between various classes of sharing opportunities.
- Evaluate the effects of sharing opportunities on hardware components such as a processor's cache hierarchy.

The chapter begins with an introduction to the evaluation methodology and an overview of the utilized hard- and software, including the configuration of the simulated machines. Afterwards, the capabilities of the proposed solution are demonstrated by presenting the results of a prototypical analysis. The chapter ends with a discussion of the findings.

5.1 Methodology

As basis for the evaluation a range of simulations with different workloads and hardware configurations were traced. The examinations take advantage of the fact that the proposed tracing-based architecture allows to perform analyzes with differing emphases on the same trace data without the need to rerun the simulations. Thus, the presented results are gathered solely by changing the analysis logic or by incorporating other data from the same traces. The evaluation is divided into two major parts.

The first analysis (Section 5.3) focuses on the classification of sharing opportunities and thereby demonstrates how the proposed approach can be utilized to get an insight into the temporal and spatial characteristics of sharing opportunities. The results are correlated to the operating system state to make statements about the semantic background of the detected sharing opportunities and to identify characteristics typical to certain types of page frames.

The second analysis (Section 5.4) is directed to examine memory duplication in a processor's cache hierarchy. For this purpose, three different cache hierarchies were simulated. The analysis is centered on a comparison of the various sharing related metrics such as the number of cache lines that could potentially be merged.

5.2 Evaluation Setup

The evaluation setup comprised two hosts that executed the simulation and performed the tracing as well as three exemplary simulation targets.

Simulation Hosts

As the simulation of a target system is a primarily single-threaded process, the simulator itself does not directly benefit from huge multi-core systems. Instead, the computational power of a single core determines the simulation speed. Thus, for the evaluation quad-core processors with a 3.4 GHz per-core frequency were used which are able to automatically overclock if the thermal state permits it. Since the compression of trace data during a simulation is performed asynchronously and fully parallelized, the tracer profits from the remaining cores. This way, the simulation is not slowed down due to the computationally costly compression. Depending on the amount of data traced, multiple simulations can be performed on a single host at the same time until all cores are fully utilized. The configuration used for the evaluation permitted to run three simulations concurrently. To minimize the overall simulation time, two systems functioned as simulation hosts for a total of six parallel simulations. Hardware and software specifications are summarized in Table 5.1.

Component	Model / Specification
Primary Host	
CPU	Intel Core i7-2600K
Cores	4 (8 logical cores)
Frequency	3.4 GHz (3.8 GHz Turbo Boost)
Memory	16 GiB (DDR3-1333)
Hard disk	2 TB Seagate Barracuda Green
Secondary Host	
CPU	Intel Core i7-2600
Cores	4 (8 logical cores)
Frequency	3.4 GHz (3.8 GHz Turbo Boost)
Memory	16 GiB (DDR3-1333)
Hard disk	500 GB Western Digital Caviar Blue
Operating System	Windows Server 2008 R2 Enterprise Edition
Architecture	64 Bit (x64)
Version	6.1.7601 Service Pack 1 Build 7601
Simulator	Wind River Simics for Windows
Architecture	32 Bit (x86)
Version	4.2.83
Extensions	Model Library Intel 440BX X86 PC 4.2.17
	Simics Model Builder 4.2.43

Table 5.1: Specification of Simulation Hosts

Simulation Targets

The core of the evaluation is built around three simple desktop workloads running on Ubuntu Linux 11.10 that comprise the execution of standard desktop software. As depicted by Figure 5.1, each workload executes a single major application such as LibreOffice or the development environment Eclipse paired with Firefox as a second "background" application. Barker et al. found such scenarios to offer a high amount of sharing opportunities which makes them well suited for prototypical analyses [10]. We do not cover any server workloads as these would have required simulating a network of at least two synchronized target systems, a server and a client. It is not feasible to stress the simulated server from a physical machine, because the simulation runs much to slow to provide timely responses. Consequently, network protocols such as TCP timeout or adapt otherwise to the high latency (e.g., adjusted window size) and thereby prevent a realistic benchmark.

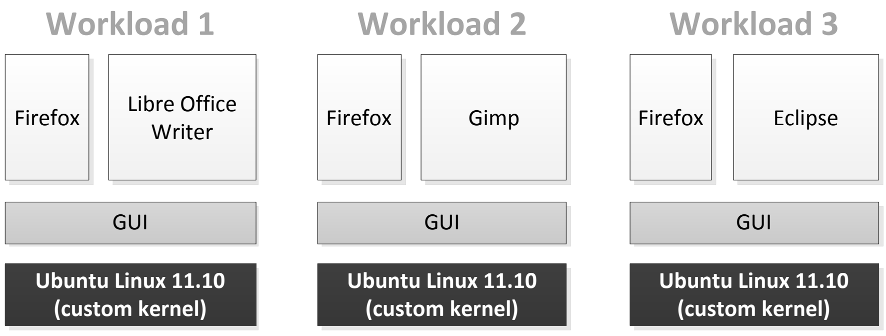

Workload 1
Firefox
Libre Office Writer
GUI
Ubuntu Linux 11.10 (custom kernel)
Workload 2
Firefox
Gimp
GUI
Ubuntu Linux 11.10 (custom kernel)
Workload 3
Firefox
Eclipse
GUI
Ubuntu Linux 11.10 (custom kernel)

Figure 5.1: Simulation Workloads

The exact software versions and information on the files used as input for the applications can be found in Table 5.2. Depending on the configuration of the simulation, the tool described in Section 4.5 executes a benchmark script that starts the selected workload.

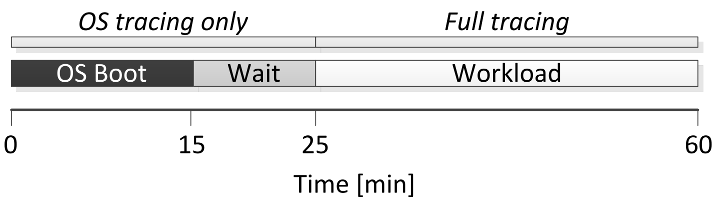


Figure 5.2: Evaluation Timeline

In every case, the benchmark script waits 10 minutes before the applications are launched to let any background activity caused by the recent boot calm down. The applications then get approximately 30 minutes to come

Component	Model / Specification
Simulated Hardware	
CPU	Intel Pentium 4
Architecture	32 Bit (x86)
Cores	1 (1 logical core)
Frequency	20 MHz (1 cycle/instruction)
Memory	2 GiB (no timing simulation)
Hard disk	8 GB System Disk
Operating System	Ubuntu Linux 11.10
Architecture	32 Bit (x86)
Kernel	Custom Vanilla Build[1] (Version 3.3.2)
Software	
Firefox	12
	www.kit.edu [local copy, 29.04.2012]
LibreOffice Writer	3.4.4 (Build 402)
	Official LibreOffice Writer Manual (13 MiB)
Gimp	2.6.11
	Picture of earth from Apollo 17[2] (6.21 MiB)
Eclipse	3.7.0 (Indigo)
	Multi-page Editor sample project

[1] Includes operating system introspection
[2] http://de.wikipedia.org/wiki/Datei:The_Earth_seen_from_Apollo_17.jpg [10.05.2012]

Table 5.2: Specification of Simulation Targets

up and load the respective input files. While operating system introspection data is traced from the beginning (which is required for a simulation replay), the tracing of sharing opportunities is activated together with the start of the workloads.

In all scenarios, the target system runs on an equally configured virtual machine. The processor is a single-core Intel Pentium 4 with 20 MHz. The low frequency was chosen so that a workload simulation time of 30 minutes could be achieved in a reasonable time frame (approx. 1.5 days). The machine is equipped with 2 GiB of main memory. For an explanation on how virtual machines are configured in Simics see Section 2.4.1.

5.3 Classification of Sharing Opportunities

The first analysis examines the spatial and temporal characteristics of the traced sharing opportunities and correlates each property with the semantic usage of the involved page frames.

The analysis is performed through a concurrent simulation replay (see Section 4.3.3) of the three gathered traces. At the same time, a query is executed on a multiplexed event stream of sharing state models to evaluate sharing group related creation/modification events. In addition, system state models deliver information about the page frame usage. This data in turn is taken to infer a sharing group's memory category (e.g. heap, file cache, etc.) Note that zero-pages are excluded from all statistics.

5.3.1 Sharing Potential

The applications selected for the individual workloads show a varying tendency to memory duplication. The temporal development of the amount of mergeable pages for each workload is depicted in Figure 5.3. With 27,000 to 30,000 pages (approx. 110 MiB), the system running the LibreOffice workload (workload 1) offers the most sharing potential. Workloads 2 (Gimp) and 3 (Eclipse) generate a comparable potential of around 22,500 to 25,000 pages (approx. 93 MiB).

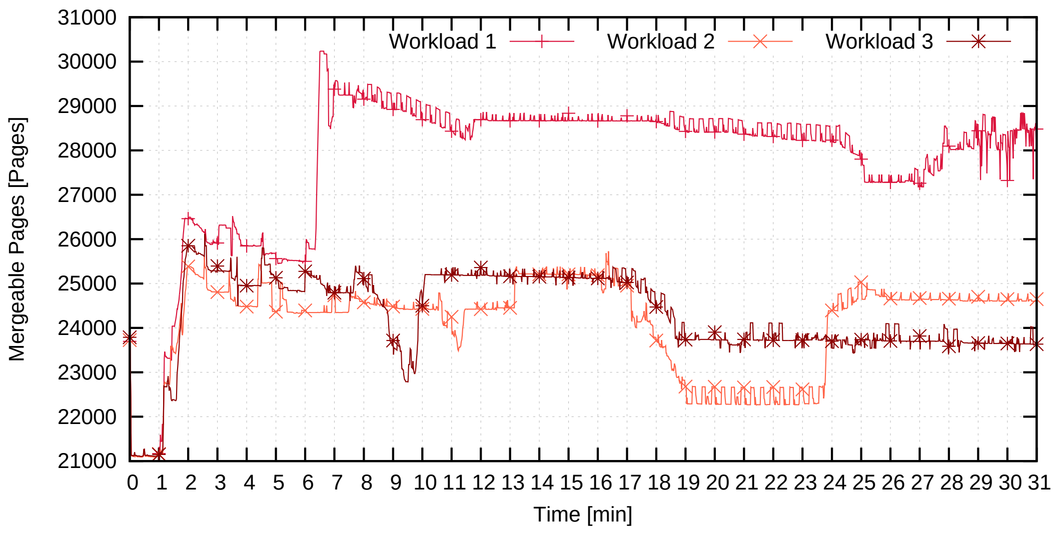


Figure 5.3: Sharing Potential by Workload. The LibreOffice workload produces the most mergeable pages.

Since the benchmark script waits 10 minutes to let the system calm down before Firefox and the respective applications are started, the number of mergeable pages at the beginning of the trace gives a good indication of the baseline sharing potential of the Ubuntu-based evaluation system. With around 80 MiB this is nearly as twice as much than Barker et al. found for a comparable 32 bit Ubuntu machine [10]. This might be due to the amount of physical memory (2 GiB) assigned in our experiments which gives plenty room for file caching. Unfortunately, Barker et al. do not state the main memory size for their VMs. With a baseline of 80 MiB, approximately 13 MiB to 30 MiB of mergeable pages originate from the test applications.

The sudden decrease in mergeable pages in the first seconds (around 10 MiB) of the trace are potentially due to memory allocations (e.g., file cache pages) required to start the new processes and caused by the zeroing of free dirty pages. As zero-pages are not counted, the sharing potential drops. As expected, the amount of mergeable pages starts to drift apart the longer the system executes different workloads. A striking detail in the temporal development is the periodic fluctuation of around 300 to 400 pages that can be observed in places especially in workload 1 and 2. The time between two consecutive changes, i.e., an increase or decrease, is nearly exactly 10 seconds, which suggests that the fluctuation is caused by a timed operation. A first examination of the processes that ran at the corresponding points in time indicates that the mergeable pages are possibly created by the kernel and are subsequently consumed by the X display manager LightDM [33] that is shipped with Ubuntu. However, it is not clear why this phenomenon is not equally visible for all workloads. Further investigations with our analysis software are necessary to clarify this behavior.

Self-Sharing vs. Inter-VM Sharing

Running memory deduplication in a virtualization host is a common practice. If the same operating system and system services run in the virtual machines, the chance of finding identical page frames is high [15, 22, 29, 55]. Barker et al. first used the terms *self-sharing* and *inter-VM sharing* to differentiate sharing opportunities that arise from identical page frames within a single VM and those that come from page frames that are mergeable between VMs only [10]. We adopt this definition throughout our evaluation and give according results for each analysis. The inter-VM sharing was calculated with the help of the multiple-trace support of the resolving

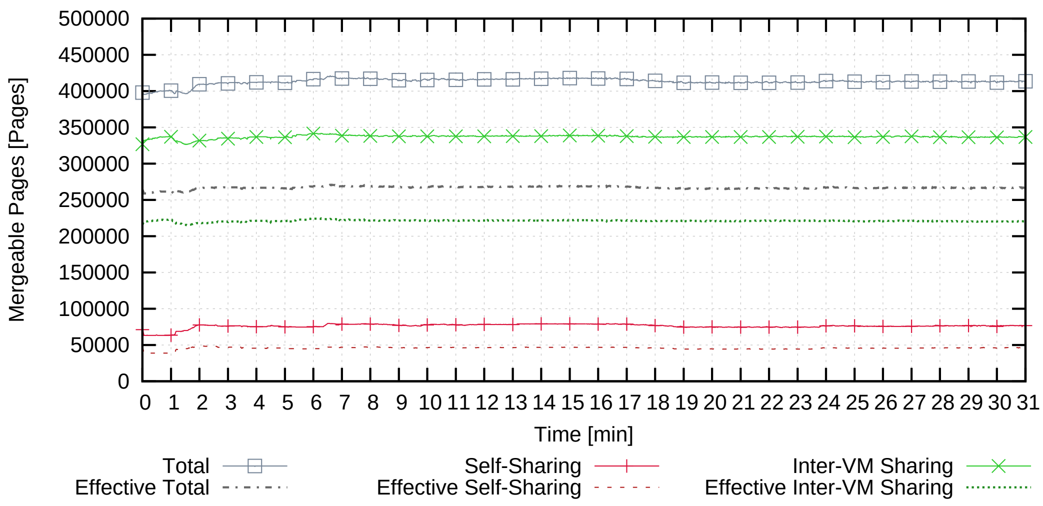


Figure 5.4: Self-Sharing vs. Inter-VM Sharing. The results are averaged over all workloads. With 81.57%, inter-VM sharing dominates the amount of mergeable pages.

mechanism (see Section 4.3.3). For both, self-sharing and inter-VM sharing, the results represent the average of the three examined workloads.

In contrast to the findings of Barker et al., our evaluation showed a high sharing potential that stems from identical page frames between the simulated systems (i.e., inter-VM sharing). Figure 5.4 depicts the ratio between self-sharing and inter-VM sharing over the time the workloads were executed. On average a total of approximately 413300 mergeable page frames (1.58 GiB of 6 GiB overall VM memory) was detected, comprising 18.43% self-sharing and 81.57% inter-VM sharing. In our evaluation the rate of inter-VM sharing benefits from the fact that the full system simulation is deterministic and therefore produces the exact same physical memory content up to the point the simulations execute different workloads. This naturally allows for a high degree of inter-VM sharing. Nevertheless, the same is true for copy-on-write based virtual machine deployment [31].

As for every set of identical page frames at least a single copy must be kept in RAM, the effective number of page frames that can be freed through memory deduplication is smaller than the plain amount of mergeable pages. Figure 5.4 also includes measurements of the number of effectively mergeable pages. While the ratio between self-sharing and inter-VM sharing roughly stays the same (17.13% vs. 82.87%), the absolute amount of total mergeable pages drops down to 266800 (approx. 1 GiB).

Correlating Sharings with Page Usage

The usage for each page frame is determined with the help of the introspection trace data and the subsequent replay of all actions (e.g., mapping of a page) in the system state model. This way, the analysis logic can easily retrieve page usage information from the OS model. A problem, however, with the correlation of sharing opportunities with specific memory categories is that sharing groups may contain page frames of many different types. Simply building a mixed category from all types of frames that a sharing group includes during its lifetime makes it hard to get a clear picture of characteristics specific to certain types of page frames. A better approach is to weight each involved page usage by the time and number of pages it contributes to a sharing group and thereby identifying a group's dominating page types. This way, the properties of a sharing group such as its rank and lifetime can be attributed to a clear memory category. To determine the memory category of a specific sharing group, the analysis logic, thus, takes the usage of each page frame that is at some point part of the group and accounts the number of CPU cycles it spends in the group. Finally, the page usage with the most cycles is chosen as the sharing group's primary memory category. If multiple categories achieve the same amount of page cycles, a combined memory category is built (e.g., file and heap). In contrast to simply counting page frames, this approach guarantees that the dominating memory categories are identified, even if individual page frames of secondary categories are often added and removed to a sharing group. This is illustrated in Figure 5.5.

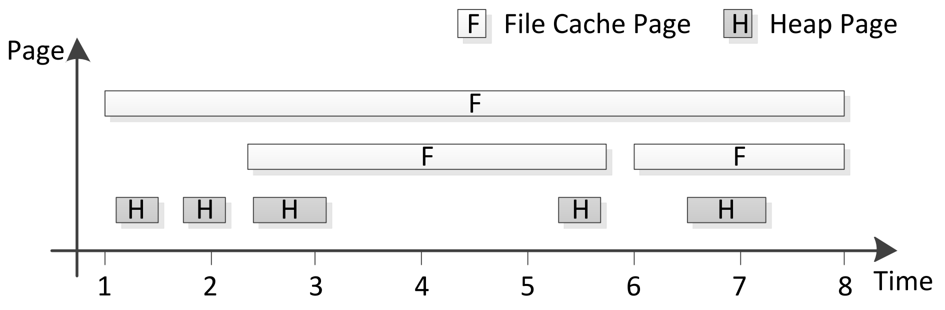


Figure 5.5: Page Cycles vs. Page Counts. Simply counting the number of page frames added and removed to a sharing group can lead to a false impression of its dominating memory category (5 heap vs. 3 file).

The results of the semantic correlation are presented in Table 5.3. With the algorithm based on page cycles, most of the sharing opportunities

Memory Category	Self-Sharing	Inter-VM Sharing	Total
File	69.78%	72.24%	71.79%
Heap	0.64%	11.14%	9.20%
Anonymous	13.52%	4.63%	6.27%
Slab Cache	0.02%	7.10%	5.80%
Reserved[1]	4.70%	3.53%	3.75%
Free and File	10.19%	0.01%	1.88%
User Stack	0.32%	0.93%	0.82%
Kernel Stack	0.00%	0.39%	0.32%
Other	0.83%	0.03%	0.18%

[1] Contains non-free pages not explicitly tracked by the OS introspection (e.g., driver private pages)

Table 5.3: Mergeable Pages by Memory Category. The top three categories are underlined. In all cases sharings that stem from the file cache are most prominent.

were originating from identical page frames in the operating system's file cache. This is the case for self-sharing (69.78%) as well as inter-VM sharing (72.24%). Heap memory and anonymous memory are also very prominent sources of sharing opportunities (9.20% total and 6.27% total, respectively). This seconds the findings of Barker et al., but stands in contrast to the results of Kloster, Kristensen, and Mejlholm who identified kernel pages as second largest source of sharing opportunities. In our evaluation kernel pages were mostly represented in the results by equal pages (inter-VM) in the slab caches (7.10%). This makes sense, as those pages are initialized with sets of equally sized objects to quickly serve frequent kernel object allocations. Looking at the results of the self-sharing, a huge amount of sharing opportunities also stems from sharing opportunities between page frames in the file cache and frames already freed by the kernel's buddy allocator (10.19%). This suggests that the free page frames were previously used in the file cache or served as temporary buffer for file contents (read or write).

The temporal development of each memory category's contribution to the total sharing potential (measured in mergeable pages) is illustrated in Figure 5.6. To preserve a clear scale, file pages are omitted. Over the period of 30 minutes, the proportion of the categories to each other roughly stays the same. The most notable change is the increase of sharing potential

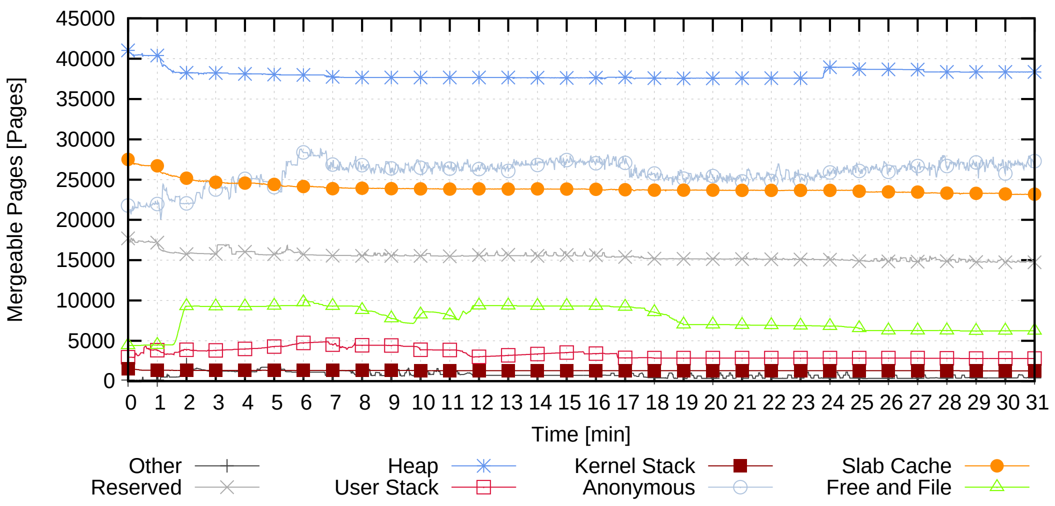


Figure 5.6: Temporal Development of Mergeable Pages by Memory Category (Total). For the evaluated workloads, the categories show only moderate variation over time.

within anonymous memory at the beginning of the trace. This is most probably caused by the increasing amount of allocated anonymous memory during the start of the workloads and the reading of the input files.

5.3.2 Rank

The rank of an individual sharing opportunity denotes the number of page frames with specific identical contents. As the rank of a sharing opportunity can vary over time, the analysis considers the time that a sharing group has a specific rank and attributes this to the respective rank category. If for instance a sharing group has a rank of 2 for 50% of the time and a rank of 3 for the other 50%, then the sharing group is equally counted (half) in both rank categories.

Miłós et al. have found 79.70% of sharing opportunities to span exactly the minimum of two page frames [38]. In their evaluation 7.01% of sharings consisted of three page frames and 13.29% involved four page frames or more. The ranks measured in our analysis lead to a slightly different picture. While the majority of sharing opportunities within a single simulation (self-sharing) also had a rank of 2 or 3, the distribution among the two categories differs. 55.24% of sharing opportunities consisted of two page frames and 38.98% spanned three page frames, which is noticeably more

Rank	Self-Sharing	Inter-VM Sharing	Total
2	55.24%	6.52%	15.44%
3	38.98%	93.46%	83.04%
4	0.23%	0.02%	0.06%
5	0.10%	0.00%	0.03%
6	0.08%	0.00%	0.02%
> 6	5.37%	0.00%	1.42%
Ø Rank	2.45	2.91	2.82

Table 5.4: Breakdown of Sharing Opportunities by Rank

than what Miłós et al. measured. This might rest on the different kinds of workloads examined (server vs. GUI/desktop). Table 5.4 summarizes our experimental results. Since our evaluation of inter-VM sharing is based on three traces, the most prominent rank for inter-VM sharing opportunities is 3. This circumstance also determines the results of the total rank distribution.

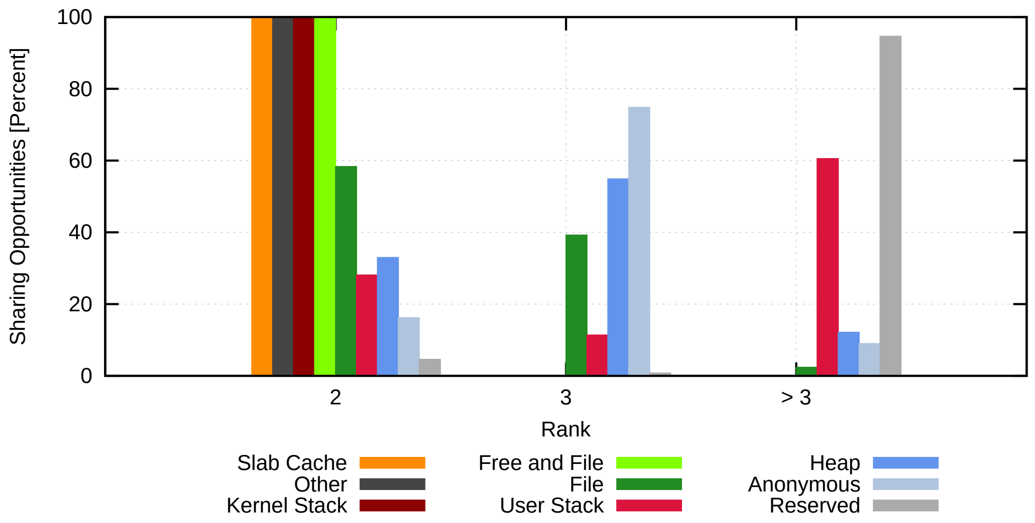


Figure 5.7: Rank by Memory Category (Self-Sharing only)

Figure 5.7 illustrates the rank distribution for the major memory categories that were identified in the last section. The averaged self-sharing over all workloads is used as data basis. The largest categories with sharing opportunities that were dominated by file (69.78%, compare Table 5.3), anonymous (13.52%) and free-and-file pages (10.19%), show a primary rank of 2 or 3. This is the reason for the clear results in Table 5.4. Sharing opportunities in the free-and-file memory category for instance have almost exclusively a rank of 2 (99.99%). Considering the categorization algorithm for sharing groups, this is a logical result. Opportunities dominated by anonymous memory pages in contrast show a typical rank of 3 (74.83%). It is striking that the semantically comparable heap category presents a very similar rank distribution with a peak at rank 3 and similar shares in the adjacent sizes.

	File	**User Stack**	**Heap**	**Anonymous**	**Reserved**
Rank	62	5	4	751	949
Percent	0.35%	19.03%	11.37%	1.76%	9.06%

Table 5.5: Primary Rank for Large Sharing Opportunities (Self-Sharing only)

Sharing opportunities that are dominated by user stack pages or reserved pages (i.e., driver private pages) are characterized through a rank greater than 3. The same is true for a small fraction of file, heap and anonymous memory based sharings. Table 5.5 reveals which rank (greater than 3) is the most prominent for each of the memory categories. Around 19% of sharing opportunities in the user stack category for instance have a rank of 5. Noticeably larger sharing opportunities can be found in the re-

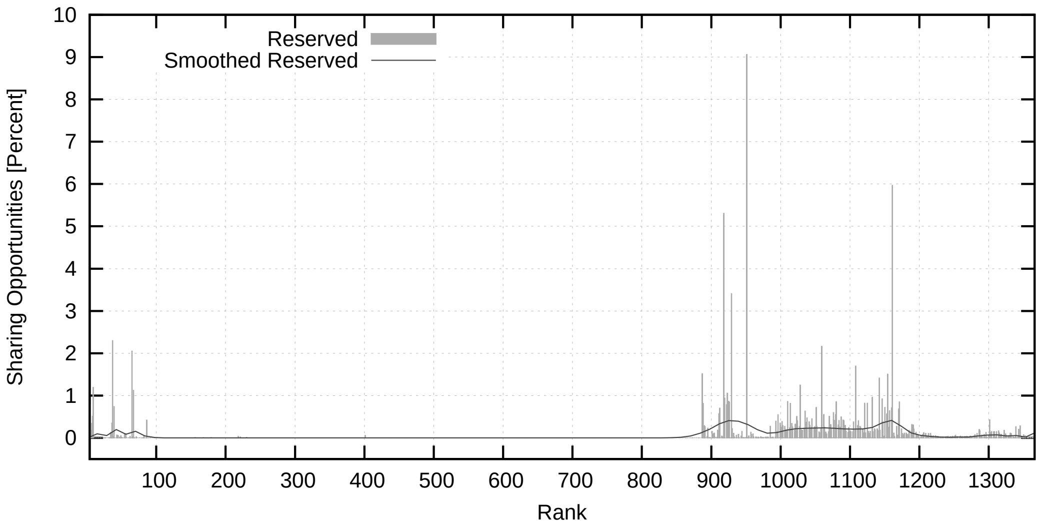


Figure 5.8: Rank Distribution for Reserved. Peaks at: 42, 70, 930, 1160

served memory category, where 9% of all sharings reach a size of 949 page frames. Figure 5.8 gives an overview of the categorie's complete rank distribution. The smoothed distribution shows peaks at 42, 70, 930 and 1160. This suggests that the category mainly covers specially initialized buffers in the range between 3.5 MiB and 4.5 MiB.

5.3.3 Lifetime

The lifetime of sharing opportunities is an important metric as merging too short-lived sharings can lead to undesirable performance penalties due to early breaking of affected copy-on-write sharings. If performance is a critical factor, it is therefore the best to merge only those sharing opportunities that provide a good trade off between rank (i.e., the amount of memory that can be freed) and longevity.

Satori is the only work that did an analysis of the temporal characteristics of sharing opportunities [38]. For a kernel build experiment with a 256 MiB virtual machine, they found that most sharings (46%) exists for a duration between 30 seconds and 5 minutes. Only 24.5% of opportunities were identified to be very short-lived with a lifetime below 30 seconds. Increasing the size of the virtual machines physical memory reduced the amount of short-lived sharing opportunities to 15.9%. The results were computed on the basis of VM memory dumps which were captured every 30 seconds. A problem with this approach is the low measurement resolution that inher-

Lifetime	**Self-Sharing**	**Inter-VM Sharing**	**Total**
$<$ 1 s	45.47%	97.00%	95.68%
$\geq$ 1 s	54.53%	3.00%	4.32%
$\geq$ 2 s	30.61%	2.38%	3.10%
$\geq$ 3 s	27.33%	2.20%	2.84%
$\geq$ 4 s	25.46%	1.96%	2.57%
$\geq$ 5 s	24.09%	1.88%	2.45%
$\geq$ 30 s	15.04%	1.25%	1.60%
$\geq$ 1 min	13.78%	1.18%	1.51%
$\geq$ 5 min	12.95%	1.12%	1.43%
$\geq$ 10 min	10.87%	1.10%	1.35%
$\geq$ 30 min	8.79%	1.02%	1.22%

Table 5.6: Integrated Lifetime Distribution

ently misses sharing opportunities that exist only in between the interval of examination. This entails the risk of distorted results.

Our simulation-based evaluation method in contrast is able to capture every sharing opportunity, even if it only exists in between two consecutive write operations. Our evaluation show that, in fact, Miłós et al. missed a large part of extremely short-lived sharing opportunities. In the case of self-sharing, over 45% of sharing opportunities only live for under a single second. With 97% this rate is even higher for inter-VM sharing. For the interval chosen in Satori this means that approximately 85% (or 98%, respectively) of sharing opportunities vanish before they reach a lifetime of 30 seconds. Table 5.6 summarizes the results. The development as well as a probability distribution of the lifetime of sharing opportunities for self- and inter-VM sharing is additionally depicted in Figure 5.9. Due to the mentioned proportions between the timeframes below and over 30 seconds, the figure omits data for the former.

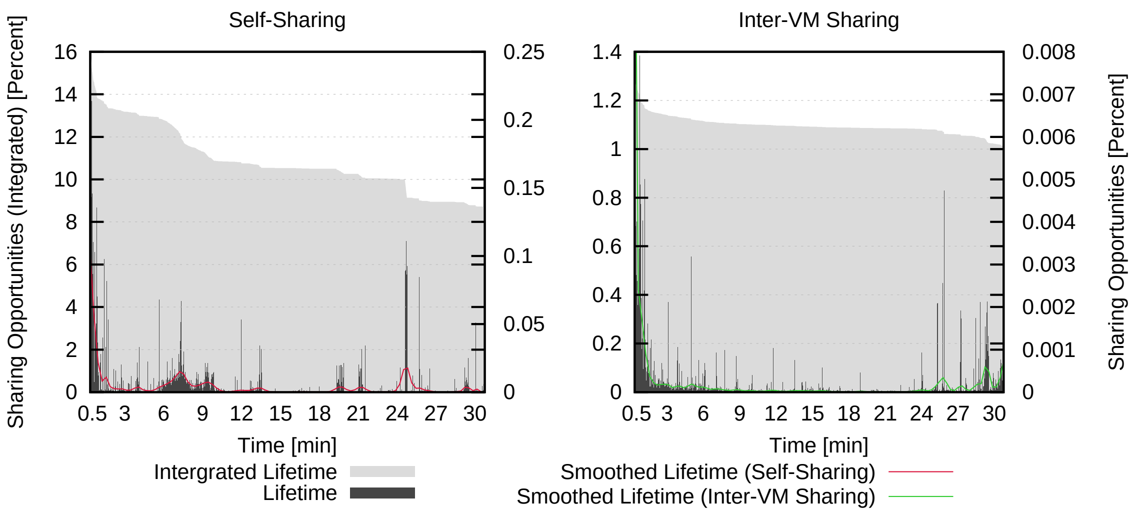


Figure 5.9: Self-Sharing vs. Inter-VM Sharing Lifetime Distribution. Starting at 30 seconds.

Examining the lifetime of sharing opportunities by memory category reveals that besides offering much sharing potential, pages in the operating system's file cache also have the longest lifetime. This makes them ideal candidates for memory deduplication. However, their lifetime naturally depends on the available physical memory which determines how long pages in the file cache can be preserved until they get evicted due to memory pressure. In addition, other OS policies can lead to an early eviction of

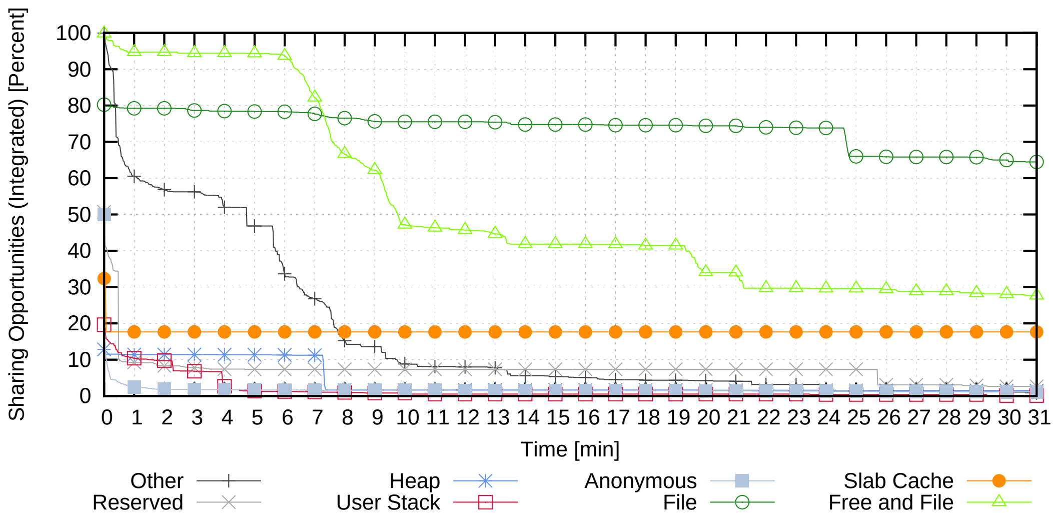


Figure 5.10: Integrated Lifetime Distribution by Memory Category (Self-Sharing only)

file pages. Figure 5.10 shows that although the evaluation system freed file cache pages, they are not reused for a long time. This suggests that the kernel evicted the file pages without experiencing memory pressure, which seems logical, considering the huge amount of physical memory assigned to the simulation (2 GiB) versus the small footprint of the chosen workloads. In contrast to the file cache pages, heap pages as well as anonymous memory pages show a noticeable shorter lifetime. In the case of the latter, only a fraction (2.42%) of sharing opportunities survives longer than one minute.

5.4 Sharing Opportunities in Caches

Memory deduplication makes it possible to run certain workloads with a smaller physical memory footprint. This raises the question if a processor's cache hierarchy can also benefit from the deduplication of memory through a reduced number of required cache lines during execution. This, in turn, would leave room to catch more memory requests.

For the evaluation of the sharing potential, cache simulations were performed and traced as part of the full system simulation. To examine the effect of memory duplication on different types and sizes of cache hierarchies, three models were simulated (see Section 2.3 for more details[1]):

- Intel Pentium 4 (Willamette) - 8-way, 256 KiB L2 cache
- Intel Core i7-2600K - 16-way, 8 MiB L3 cache
- Intel Xeon E52470 - 20-way, 20 MiB L3 cache

In each case only the last cache level was analyzed as this is typically the largest one and, thus, offers the most potential to contain cache lines that reference distinct page frames with equal contents. The term *mergeable cache lines* is used to denote this type of lines.

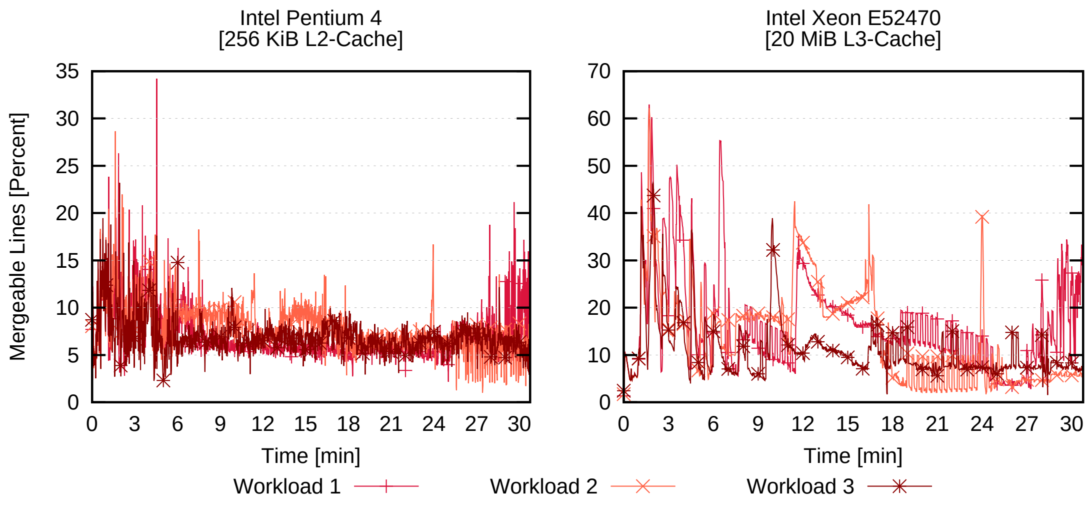


Figure 5.11: Mergeable Cache Lines (Self-Sharing only)

Figure 5.11 depicts the temporal development of the amount of mergeable cache lines for each workload over the time of simulation. The measurements show that depending on the cache size, the development of the number of mergeable pages as illustrated in Figure 5.3 is reflected in the cache. Even the fluctuations, for instance in workload 1 in the range between minute 18 and 24, are clearly visible in the Xeon's profile. The measurements for the Pentium 4 show that with decreasing size, the cache only approximates such patterns. This makes sense as the limited capacity forces the cache to frequently evict cache lines in favor of new ones.

[1]The Xeon's cache hierarchy only differs from the Core i7's ones in the associativity and size of the L3 cache.

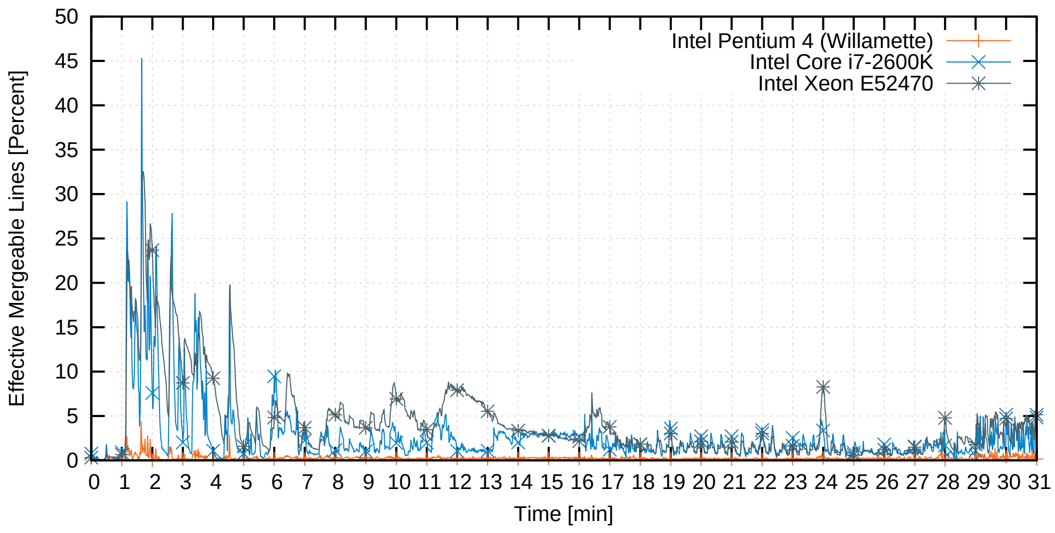


Figure 5.12: Effective Mergeable Cache Lines (Self-Sharing only)

Nevertheless, even for small caches a clear relationship between the number of mergeable pages and the amount of mergeable cache lines can be proven. On average, 7.26% of the lines in the Pentium 4's L2 cache and 13.81% of lines in the Xeon's L3 cache reference mergeable pages.

Since not all of the page frames to which the cache points to can be freed (a single copy per contents must be preserved), the amount of cache lines that can potentially be saved is smaller than the plain number of mergeable cache lines. Figure 5.12 illustrates the percentage of each cache that can be freed if this is taken into account. These results do not consider each cache line's offset into its page frame. Instead, the results are inferred by calculating the average number of cache lines per referenced page frame and multiplying this value with the amount of surplus referenced page frames (due to identical contents). This approximation thus assumes that for each page frame the corresponding lines point to the same offsets and can therefore be freed. Obviously, this is the best-case and gives only an indication of the maximum potential. In the worst-case, on the other hand, every line references a different offset within identical frames and hence needs to be preserved. In the consequence, no lines can be freed. However, the worst-case is limited due to the fact that for instance for a cache line size of 64 byte only 64 cache lines may reference different offsets in a 4 KiB target frame. Each extra cache line (that references a different but identical frame) is redundant. The worst case, therefore, depends on the cache organization and size.

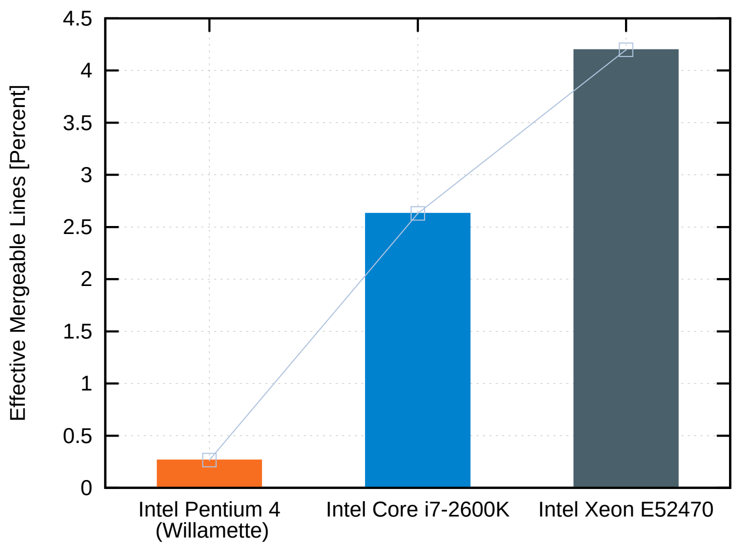


Figure 5.13: Average Effective Mergeable Cache Lines (Self-Sharing only)

Figure 5.13 contrasts the average percent of cache lines that can effectively be freed in the best-case. As apparent, larger caches show noticeable more potential to benefit from memory deduplication. Moreover, increasing a cache's size leads to a superlinear increase in sharing potential. While the Core i7's cache is only 64 times larger than the Pentium 4's one, it contains about 100 times more mergeable pages. Transitioning from the Core i7 to the Xeon shows a similar development, however, the discrepancy is less in this case (factor 1.2 vs. 1.6).

5.5 Conclusion

The evaluation showed that the proposed simulation-based analysis mechanism is capable to give a detailed insight into the characteristics of sharing opportunities. Moreover, it is able to correlate the findings with the operating system state.

For the examined workloads, file cache pages could be identified as a valuable target for memory deduplication. They combine a large sharing potential (approx. 70% of all sharings) with the highest lifetime (65% survive longer than 30 minutes) and are therefore ideal candidates for sharing. This seconds the findings of previous research [10, 29, 38]. With around 15% of all sharing opportunities, anonymous and heap pages also offer a high sharing potential. However, they typically show a much shorter lifetime than pages within the file cache.

The evaluation also illustrated that without the analysis granularity provided in this work, measurements may get distorted. This is the case with the results presented by Miłós et al. concerning the lifetime of sharing opportunities [38]. Their evaluation missed most of the very short-lived sharing opportunities and, thus, they did not find that 95% of sharings exist for less than a single second.

With regard to the ratio between self-sharing and inter-VM sharing, a remarkable difference to the results of Barker et al. could be determined. While Barker et al. found that the majority of sharing potential stems from self-sharing [10], our evaluation did not confirm this. Even more, with on average 81% inter-VM sharing, our results indicate the contrary. Further investigation is needed, to clarify if this discrepancy originates from the differing measurement methods (virtual machine memory dumps vs. comparison of simulation traces).

A first examination of sharing potential in caches revealed that especially the large caches of server processors such as the 20 MiB L3 cache of the Intel Xeon E52470 provide the potential to benefit from memory deduplication. In contrast to small caches (e.g., the 256 KiB Pentium 4 L2 cache) that suffer from too frequent cache line eviction, large caches may save on average over 4% of cache lines for a deduplicated workload.

Chapter 6

Conclusion

Memory duplication occurs when multiple page frames hold identical contents, thereby introducing unnecessary data redundancy in main memory. Memory deduplication aims at reducing this redundancy by copy-on-write remapping page frames with equal contents to a single copy in RAM. It is important to have a thorough knowledge of the characteristics of such sharing opportunities to focus deduplication efforts to promising memory areas and to avoid deduplicating page frames that lead to wasted computational overhead caused by the early breaking of established sharings. However, previous research leaves many open questions regarding the temporal and spatial properties of sharing opportunities as well as regarding the correlation of these characteristics to operating system state information (e.g., processes, address space areas, page usage, etc.). Moreover, no evaluation exists that deals with the effects of memory deduplication on the performance of hardware components such as processor caches.

In this work, a framework has been proposed that combines fine granular continuous tracing with a flexible and extensible analysis concept to answer these questions. In contrast to previous work, that for the most part based its examinations on virtualization and sampling, our approach employs full system simulation as data acquisition environment. The binary translation that comes with this method enables our approach to provide temporal completeness of information regarding memory operations, sharing opportunities and operating system operations. The respective information is gathered through memory inspection and operating system introspection. A lightweight register-based communication channel has been presented that allows transferring introspection data with minimal overhead and side-effects for the simulated system. This lays the foundation for an analysis of sharing opportunities with maximum resolution. In fact, our evaluation showed that some of the results presented by previous work are distorted due to the low data resolution inherent to sampling.

The applicability of the proposed tool chain has been demonstrated with an evaluation of sharing opportunities for prototypical desktop workloads. The evaluation illustrated that the presented analysis mechanism is capable to give a detailed insight into the characteristics of sharing opportunities. Moreover, for the first time an examination of the sharing potential in processor caches has been performed that shows that memory deduplication can increase cache performance by reducing the number of cache lines required to run the same workload.

6.1 Future Work

Our primary focus for the future lies in improving the proposed tool chain to conduct a more advanced evaluation based on real world scenarios. We especially plan to examine server workloads such as a typical LAMP stack or virtualization-based server consolidation as these are common targets for memory deduplication. To this end, a major challenge will be to enable the full system simulation of such complex configurations in a reasonable timeframe. This might include the adaption of the proposed tracer to a simulation platform that supports processor models with instructions for hardware virtualization.

An area not covered by the evaluation is the analysis of memory access patterns. Determining any characteristics in this field may help to improve the effectiveness of memory deduplication through optimized timing (e.g., wait 2 minutes after a page is mapped until its memory is scanned). Since the proposed analysis software offers for every of the interesting events, such as mappings or CPU writes, corresponding event streams to perform queries on, an examination of memory access patterns can easily be done. With this technique, we already made the observation that a surprisingly high share of write operations (around 34%) does effectively not change a frame's contents as the target memory already holds the intended data. Since these types of writes do not require a COW break for deduplicated frames, we call this type *non-destructive writes*. We plan to invest further research into this direction.

Our first evaluation results concerning the sharing potential within processor caches show that this area is also a promising target for future research. We therefore intend to perform more thorough cache analyses that take each cache line's offset into account and thereby allow making more sound statements about the effective sharing potential. Moreover, we plan to measure actual cache related performance differences resulting from memory deduplication and investigate if modifications to, for instance, the operating system's process scheduling can increase potential performance benefits.

Another direction for future work is the examination of memory deduplication effects in NUMA systems. Current deduplication techniques deduplicate page frames without take NUMA nodes into account and may thus degrade memory read performance for remote nodes. As NUMA systems are becoming increasingly popular today, further research in this field is required.

Appendix A

Page Cache Files

The file cache has been found to be the most prominent source of sharing opportunities. The following tables list the top twenty of the files with the most sharing potential (measured in percent of total page cycles).

Rank	File Name	Cycles
1	/var/cache/apt/pkgcache.bin.o6NBEA	24.33%
2	/var/cache/apt/srcpkgcache.bin	24.14%
3	/usr/lib/firefox/libxpcom.so	13.99%
4	/usr/lib/jvm/java-6-openjdk/jre/lib/i386/client/classes.jsa	4.37%
5	/usr/lib/libreoffice/basis3.4/program/libswli.so	2.25%
6	/lib/i386-linux-gnu/libc-2.13.so	1.74%
7	/usr/lib/jvm/java-6-openjdk/jre/lib/i386/client/libjvm.so	1.52%
8	/usr/lib/eclipse/.../org.eclipse.jdt.ui_3.7.0.dist.jar	1.36%
9	/usr/lib/libreoffice/basis3.4/program/libsvxcoreli.so	1.30%
10	/usr/lib/firefox/omni.ja	1.14%
11	/usr/lib/jvm/java-6-openjdk/jre/lib/rt.jar	1.04%
12	/usr/lib/libreoffice/basis3.4/program/libvclli.so	0.94%
13	/usr/lib/eclipse/.../org.eclipse.jdt.core_3.7.0.dist.jar	0.83%
14	/usr/lib/libreoffice/basis3.4/program/libsvtli.so	0.78%
15	/usr/lib/libreoffice/basis3.4/program/libxoli.so	0.71%
16	/usr/lib/libreoffice/basis3.4/program/libsfxli.so	0.69%
17	/usr/share/icons/hicolor/icon-theme.cache	0.66%
18	/usr/lib/libreoffice/basis3.4/program/libpackage2.so	0.64%
19	/usr/lib/libreoffice/basis3.4/program/libtkli.so	0.62%
20	/usr/lib/libreoffice/basis3.4/program/libfwili.so	0.56%

Table A.1: Files by Cycles of Mergeable Pages (Self-Sharing)

Rank	File Name	Cycles
1	/usr/lib/firefox/libxpcom.so	11.48%
2	/usr/lib/i386-linux-gnu/libQtGui.so.4.7.4	8.73%
3	Linux Kernel Image	6.99%
4	/usr/lib/libgtk-3.so.0.200.0	4.97%
5	/usr/lib/i386-linux-gnu/libgtk-x11-2.0.so.0.2400.6	4.62%
6	/usr/share/icons/hicolor/icon-theme.cache	4.15%
7	/usr/lib/jvm/java-6-openjdk/jre/lib/i386/client/classes.jsa	2.81%
8	/usr/lib/firefox/omni.ja	2.78%
9	/usr/lib/i386-linux-gnu/libQtXmlPatterns.so.4.7.4	2.01%
10	/usr/share/icons/gnome/icon-theme.cache	1.99%
11	/usr/lib/python2.7/dist-packages/gtk-2.0/gtk/_gtk.so	1.79%
12	/usr/lib/libapt-pkg.so.4.11.0	1.22%
13	/var/cache/apt/srcpkgcache.bin	1.15%
14	/usr/lib/locale/locale-archive	1.08%
15	/usr/lib/sse2/libxapian.so.22.3.0	1.01%
16	/lib/i386-linux-gnu/libc-2.13.so	0.94%
17	/var/cache/apt/pkgcache.bin.o6NBEA	0.92%
18	/usr/lib/jvm/java-6-openjdk/jre/lib/i386/client/libjvm.so	0.87%
19	/usr/share/fonts/truetype/ttf-dejavu/DejaVuSans.ttf	0.77%
20	/lib/i386-linux-gnu/libdbus-1.so.3.5.7	0.74%

Table A.2: Files by Cycles of Mergeable Pages (Inter-VM Sharing)

Rank	File Name	Cycles
1	/usr/lib/firefox/libxpcom.so	12.70%
2	/var/cache/apt/srcpkgcache.bin	12.32%
3	/var/cache/apt/pkgcache.bin.o6NBEA	12.30%
4	/usr/lib/i386-linux-gnu/libQtGui.so.4.7.4	4.51%
5	Linux Kernel Image	3.60%
6	/usr/lib/jvm/java-6-openjdk/jre/lib/i386/client/classes.jsa	3.57%
7	/usr/lib/libgtk-3.so.0.200.0	2.56%
8	/usr/share/icons/hicolor/icon-theme.cache	2.46%
9	/usr/lib/i386-linux-gnu/libgtk-x11-2.0.so.0.2400.6	2.38%
10	/usr/lib/firefox/omni.ja	1.98%
11	/lib/i386-linux-gnu/libc-2.13.so	1.33%
12	/usr/lib/jvm/java-6-openjdk/jre/lib/i386/client/libjvm.so	1.18%
13	/usr/lib/libreoffice/basis3.4/program/libswli.so	1.09%
14	/usr/share/icons/gnome/icon-theme.cache	1.04%
15	/usr/lib/i386-linux-gnu/libQtXmlPatterns.so.4.7.4	1.03%
16	/usr/lib/python2.7/dist-packages/gtk-2.0/gtk/_gtk.so	0.92%
17	/usr/lib/jvm/java-6-openjdk/jre/lib/rt.jar	0.79%
18	/usr/lib/eclipse/.../org.eclipse.jdt.ui_3.7.0.dist.jar	0.66%
19	/usr/lib/libreoffice/basis3.4/program/libsvxcoreli.so	0.63%
20	/usr/lib/libapt-pkg.so.4.11.0	0.63%

Table A.3: Files by Cycles of Mergeable Pages (Total Sharing)

Bibliography

[1] *2nd Generation Intel Core Processor Family Desktop, Intel Pentium Processor Family Desktop, and Intel Celeron Processor Family Desktop Datasheet*. Intel Corporation, Dec. 2011.

[2] *7-zip SDK*. URL: `http://www.7-zip.de/sdk.html`.

[3] 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*. ASPLOS-XII. San Jose, California, USA: ACM, 2006, pp. 2–13.

[4] *Advanced Features of Simics*. Wind River Systems Inc. Nov. 2011.

[5] *Advanced Memory and Timing in Simics*. Wind River Systems Inc. Nov. 2011.

[6] *AMD64 Virtualization Codenamed “Pacifica” Technology - Secure Virtual Machine Architecture Reference Manual*. Advanced Micro Devices. May 2005.

[7] Andrea Arcangeli, Izik Eidus, and Chris Wright. “Increasing memory density by using KSM.” In: *Proceedings of the Linux Symposium*. July 2009.

[8] Yungang Bao et al. “HMTT: a platform independent full-system memory trace monitoring system.” In: *Proceedings of the 2008 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*. SIGMETRICS ’08. ACM, 2008, pp. 229–240.

[9] Paul Barham et al. “Xen and the art of virtualization.” In: *Proceedings of the nineteenth ACM Symposium on Operating systems principles (SOSP)*. ACM Press, Oct. 2003, pp. 164–177.

[10] Sean Barker et al. “An empirical study of memory sharing in virtual machines.” In: *Proceedings of the USENIX ATC*. USENIX Association, 2012.

[11] Fabrice Bellard. “Qemu, a fast and portable dynamic translator.” In: *Proceedings of the USENIX ATC*. USENIX Association, 2005.

[12] *Bochs - A IA-32 Emulator*. URL: `http://bochs.sourceforge.net/`.

[13] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel, Third Edition*. O’Reilly Media, 2006.

[14] Edouard Bugnion et al. “Disco: Running Commodity Operating Systems on Scalable Multiprocessors.” In: *ACM Transactions on Computer Systems (TOCS)*. Nov. 1997.

[15] 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*. 2011.

[16] David Cheriton et al. “HICAMP: architectural support for efficient concurrency-safe shared structured data access.” In: *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2012.

[17] Jan Edler and Mark Hill. *DineroIV*. University of Wisconsin. URL: `http://pages.cs.wisc.edu/~markhill/DineroIV/`.

[18] Jakob Engblom. *Simics and Multicore Systems Development*. White Paper. Virtutech, Mar. 2009.

[19] *Gleipnir*. University of North Texas, Computer Systems Research Laboratory. URL: `http://csrl.cse.unt.edu/gleipnir/`.

[20] Ruth E. Goldenberg, Denise E. Dumas, and Saro Saravanan. *OpenVMS Alpha Internals: Scheduling and Process Control*. Digital Press, 1996.

[21] Kinshuk Govil et al. “Cellular disco: resource management using virtual clusters on shared-memory multiprocessors.” In: *ACM Transactions on Computer Systems (TOCS)*. Aug. 2000.

[22] Diwaker Gupta et al. “Difference Engine: Harnessing Memory Redundancy in Virtual Machines.” In: Communications of the ACM, Oct. 2010.

[23] *Intel 64 and IA-32 Architectures Software Developer’s Manual - Volume 2A: Instruction Set Reference, A-L*. Intel Corporation. Oct. 2011.

[24] *Intel 64 and IA-32 Architectures Software Developer's Manual - Volume 2B: Instruction Set Reference, M-Z*. Intel Corporation. Oct. 2011.

[25] *Intel 64 and IA-32 Architectures Software Developer's Manual - Volume 3B: System Programming Guide, Part 2*. Intel Corporation. Oct. 2011.

[26] *Intel 64 and IA-32 Architectures Software Developer's Manual - Volume 3C: System Programming Guide, Part 3*. Intel Corporation. Oct. 2011.

[27] *Intel Pentium 4 Processor in the 423-pin Package at 1.30, 1.40, 1.50, 1.60, 1.70, 1.80, 1.90 and 2 GHz Datasheet*. Intel Corporation, Aug. 2001.

[28] Avi Kivity et al. "kvm: the Linux Virtual Machine Monitor." In: *Proceedings of the Linux Symposium*. June 2007.

[29] Jacob Faber Kloster, Jesper Kristensen, and Arne Mejlholm. *Determining the use of Interdomain Shareable Pages using Kernel Introspection*. Tech. rep. Aalborg University, Department of Computer Science, June 2007.

[30] Ricardo Koller and Raju Rangaswami. "I/O Deduplication: Utilizing Content Similarity to Improve I/O Performance." In: *ACM Transactions on Storage*. Sept. 2010.

[31] H. Andrés Lagar-Cavilla et al. "SnowFlock: Rapid Virtual Machine Cloning for Cloud Computing." In: *Proceedings of the fourth ACM European conference on Computer systems*. 2009.

[32] *Language Integrated Query (LINQ)*. URL: `http://msdn.microsoft.com/en-us/library/bb397926.aspx`.

[33] *LightDM*. URL: `http://www.freedesktop.org/wiki/Software/LightDM`.

[34] Peter Magnusson and Bengt Werner. "Efficient Memory Simulation in SimICS." In: *In Proceedings of the 28th Annual Simulation Symposium*. 1995, pp. 62–73.

[35] Peter S. Magnusson et al. "Simics: A Full System Simulation Platform." In: *Computer* 35.2 (Feb. 2002).

[36] *Microsoft .Net Framework*. URL: `http://www.microsoft.com/germany/net/net-framework.aspx`.

[37] Konrad Miller et al. *KSM++: Using I/O-based hints to make memory-deduplication scanners more efficient*. 2012.

[38] Grzegorz Miłós et al. “Satori: Enlightened page sharing.” In: *Proceedings of the USENIX ATC*. USENIX Association, 2009.

[39] *Modeling Your System in Simics*. Wind River Systems Inc. Nov. 2011.

[40] David Mosberger and Tai Jin. “httperf - a tool for measuring web server performance.” In: *SIGMETRICS Performance Evaluation Review* (Dec. 1998).

[41] Raphael Neider. *The OpenProcessor Platform: Fostering Research on the Hardware/Software Boundary*. Tech. rep. 1. Karlsruhe Institute of Technology, Germany, Jan. 2011.

[42] Gil Neiger et al. “Intel Virtualization Technology: Hardware support for efficient processor virtualization.” In: *Intel Technology Journal* 10.3 (Aug. 2006).

[43] Mark Russinovich and Bryce Cogswell. *VMMap process memory analysis utility*. May 2011. URL: `http://technet.microsoft.com/en-us/sysinternals/dd535533`.

[44] Mark E. Russinovich and David A. Solomon. *Windows Internals, Fifth Edition - Covering Windows Server 2008 and Windows Vista*. Microsoft Press, June 2009.

[45] Martin Schwidefsky et al. “Collaborative Memory Management in Hosted Linux Environments.” In: *Proceedings of the Linux Symposium*. Vol. 2. 2006.

[46] Oleg Shilo. *CS-Script - The C# Script Engine*. URL: `http://www.csscript.net/`.

[47] Abraham Silberschatz, Peter Baer Galvin, and Greg Gagne. *Operating System Concepts*. John Wiley Sons.

[48] *Simics Feature List*. Wind River Systems Inc. Nov. 2011.

[49] *Simics Reference Manual*. Wind River Systems Inc. Nov. 2011.

[50] Mani B. Srivastava, Anantha P. Chandrakasan, and R. W. Brodersen. “Predictive system shutdown and other architectural techniques for energy efficient programmable computation.” In: *IEEE Trans. Very Large Scale Integr. Syst.* 4.1 (Mar. 1996), pp. 42–55.

[51] *Valgrind*. URL: `http://valgrind.org/`.

[52] *Virtual Box*. Oracle. URL: `https://www.virtualbox.org/`.

[53] *VMware VMmark 2.0*. VMware, Inc. URL: `http://www.vmware.com/products/vmmark/`.

[54] Michael Vrable et al. “Scalability, Fidelity, and Containment in the Potemkin Virtual Honeyfarm.” In: *Proceedings of the 20th Symposium on Operating System Principles (SOSP)*. 2005.

[55] Carl A. Waldspurger. “Memory Resource Management in VMware ESX Server.” In: *Proceedings of the fifth Symposium on Operating Systems Design and Implementation (OSDI)*. Dec. 2002.

[56] Andrew Whitaker, Marianne Shaw, and Steven D. Gribble. “Denali: Lightweight Virtual Machines for Distributed and Networked Applications.” In: *In Proceedings of the USENIX Annual Technical Conference*. 2002.

[57] *Wind River Simics*. Wind River Systems Inc. URL: `http://www.windriver.com/products/simics/`.

[58] Hui Zeng et al. “MPTLsim: a cycle-accurate, full-system simulator for x86-64 multicore architectures with coherent caches.” In: *SIGARCH Comput. Archit. News* 37.2 (July 2009), pp. 2–9.