Karlsruher Institut für Technologie

Analyzing Duplication in Incremental High Frequency Checkpoints

Bachelorarbeit
von

Jan Ruh

an der Fakultät für Informatik

Erstgutachter:	Prof. Dr. Frank Bellosa
Betreuender Mitarbeiter:	Dipl.-Inform. Marc Rittinghaus

Bearbeitungszeit: 07. Juni 2015 – 06. September 2015

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Karlsruhe, den 7. September 2015

Deutsche Zusammenfassung

Die Simulation vollständiger Computersysteme (Full System Simulation) leidet unter einer schlechten Ausführungsgeschwindigkeit, so dass sie für die Anwendung auf zeitintensive Arbeitslasten (Workloads) ungeeignet ist. Um dieses Problem zu lösen stellt SimuBoost eine parallele, verteile Simulation eines Computersystems, basierend auf hochfrequenten Checkpoints einer virtuellen Maschine, zur Verfügung. Eicher [8] stellt einen schnellen inkrementellen Checkpointmechanismus mit Deduplikation zur Verfügung, welcher für die Anwendung mit SimuBoost geeignet ist.

Die Auswertung von inkrementellen Checkpoints hat ergeben, dass viele Speicherseiten und Disksektoren mehrfach auftreten. Da inkrementelle Checkpoints nur Speicherseiten und Disksektoren in Betracht ziehen die sich verändert haben, stellt sich die Frage welchen Ursprung und welchen semantischen Hintergrund diese Duplikate haben. Zu diesem Zweck wird eine Analyse des kompletten Systems durchgeführt (Full System Analysis) um relevante Daten der virtuellen Maschine zu sammeln und auszuwerten.

Die Auswertung hat gezeigt, dass zwischen 30% und 60% der doppelten Speicherseiten unbenutzt sind. Es wurde festgestellt, dass zwischen 40% und 50% der Duplikate ihren Ursprung in ausschließlich Null enthaltenden Speicherseiten haben.

Abstract

Full system simulation suffers from a bad execution speed making it unsuitable for application with long-running workloads. To solve this execution speed issue, SimuBoost provides simultaneous full system simulation based on high frequency virtual machine checkpoints. Eicher [8] provides a fast checkpointing mechanisms, using incremental checkpoints with deduplication for the application with SimuBoost.

Evaluation of incremental checkpoints showed high duplication potential of page frames and disk sectors. As incremental checkpoints only consider dirty page frames and disk sectors, the question arises from which sources these recurring dirty pages stem from and what semantic background they have. For this purpose it uses full system analysis to collect relevant data from a virtual machine and evaluates it.

The evaluation shows that between 30% and 60% of dirty and duplicate page frames in incremental checkpoints are unused. It is found that between 40% and 50% of duplicates have their origin in page frames that contain only zeroes.

Contents

Chapter 1

Introduction

Full system simulation is a valuable tool for system analysis. Unfortunately it suffers from a bad execution speed making it almost unsuitable for analysis of time consuming workloads. Rittinghaus et al. [17] developed a full system simulation solution that emulates a workload simultaneously on mutliple nodes. It maintains one utilized node using virtualization to fast-forward system state and taking periodical checkpoints, which are distributed to simulation nodes. As the performance advantage by the simultaneous simulation highly depends on an efficient and fast checkpoint mechanism Eicher [8] presents a suited solution in his thesis, combining incremental checkpoints and deduplication with asynchronous storage.

Previous work on incremental checkpoints with deduplication by Baudis [5] showed a significantly high rate of page frame and disk sector duplicates, indicating recurrent memory content. These results are confirmed by Eicher [8]. As incremental checkpoints only consider dirty page frames and disk sectors, the question arises from which sources these recurring dirty pages stem from and what semantic background they have. Detailed information about duplicate page frames could be used to further improve the performance of incremental checkpoints. However, there is no research describing characteristics of duplicate page frames in incremental checkpoints.

The objective of this thesis is to provide a survey of page frame characteristics in incremental checkpoints and to analyze duplicate page frames in order to find prospects for extensions or optimizations for incremental checkpointing mechanisms.

The inspected system runs in its own context inside a virtual machine (VM). To analyze page frame characteristics of the operating system that is encapsulated in the VM, the so-called semantic gap has to be bridged. In the thesis at hand this is achived by using operating system introspection, which actively sends operating system events that can be used to reassemble page frame semantics to the simulation host. The host processes received data and persists it for further analysis.

An analysis tool is used to create a model describing the page frame semantics by sequentially applying operating system events from the storage. By evaluating statistics derived from the model, it is found that between 30% and 60% of dirty and duplicate page frames in incremental checkpoints are unused. It is found that between 40% and 50% of duplicates have their origin in page frames that contain only zeroes.

Chapter 2 writes of full system simulation and their field of application. It covers past work regarding the analysis of main memory duplicates and full system analysis in general. Chapter 3 gives a definition on memory semantics and analyzes requirements to solve the problem. Chapter 4 explains how the semantic gap is closed and provides an overview of the intended data analysis. In Chapter 5 is described how data is collected, transfered to the host and stored for later analysis. It also describes the implementation of the analysis tool used to evaluate collected data. Chapter 6 evaluates collected data and presents the results of this thesis. Chapter 7 concludes the thesis and gives an outlook on future work.

Chapter 2

Background

This chapter covers the fundamentals of full system simulation and incremental checkpointing. It also explains and classifies already existing work regarding memory analysis of operating systems and analysis of page frame duplicates.

2.1 Full System Simulation

Full system simulation provides developers and researchers with a useful tool to take a deeper look on system internal behavior. It empowers them to install and run unmodified operating systems (OS) in a self-contained, observable environment and make predictions and propositions about real systems. To enable the user in doing so, full system simulators reproduce the behavior of real hardware including processors, memory, peripheral devices, video and network interfaces [9].

The level of abstraction at which real hardware is simulated depends on the specific requirements of the experiment. For system verification purposes or early software development, when real hardware is not available yet, a functional full system simulation is sufficient as it guarantees correct behavior of the systems components. If accurate timing is needed additionally e.g., to evaluate the performance of a system, a timing simulation is used as it assures correct time ordering of generated functional events [3].

Full system simulators such as Simics [14] or QEMU [6] certainly can also be used to run a virtual machine (VM). Still they differ from virtual machine monitors, which are using specialized hardware extensions to achive a better performance in comparison to full system simulators, which lack in performance but emulate the hardware in more detail. The slowdown by emulating a system is 33 times compared to a virtualized solution. If the simulated system is inspected additionally for analysis purposes, the slowdown increases to 165 times compared to a virtualization [17].

2.1.1 QEMU

QEMU is an open source full system simulator, capable of interfacing with the linux kernel-based virtual machine (KVM), which executes on several architectures. It runs unmodified OS versions and comes with debugging and system inspection facilities [18, 19]. In cooperation with KVM it uses virtualization extensions of the processor manufacturers to provide an execution speed comparable to real hardware. In this case QEMU runs as a userspace process emulating devices and using, if available, KVM which exposes the hardware acceleration functions of the CPU from the kernelspace to the userspace.

However, originally QEMU was designed to execute code of an external CPU architecture. Therefore it is able to emulate well-established CPU architectures like x86, ARM or MIPS and more exotic ones[1]. To offer this wide range of emulation targets QEMU uses a dynamic code translator called Tiny Code Generator (TCG). TCG translates guest code up to the next jump or virtual CPU state changing instruction. This basic blocks of code are called Translated Blocks (TBs). QEMUs internal cache holds the most recently used TBs for faster translation. Each target CPU instruction of a block splits into a couple of TCG operations that get translated into host CPU instructions [20]. A host instruction modifies the guests system state as intended by the original guest operation.

There are guest instructions, which are not translated into host instructions but rather handled by so-called helper functions. QEMU uses helper functions to implement uncommon but complex architecture specific instructions so they do not have to get implemented as large code blocks that are translated during runtime, therefore affecting the execution speed negatively.

With the use of TCG, QEMU archives a better performance compared to other full system simulators. Running a Linux kernel build QEMU is 24 times faster than Simics. Even with analysis functionality activated QEMU is still 5 times faster than Simics [17].

The speedup compared to other full system simulators is achived by trading simulation precission and timing for execution speed benefits. Nevertheless QEMU is sufficient to analyze for example memory access patterns or the operating system state. Still if a more precise simulation of the hardware is required QEMU is of no more use and more accurate simulators, that lack in passable performance, must be employed.

[1]For a full list, see [18]

2.2 SimuBoost

Existing full system simulators lack in performance, as mentioned before. They either simulate a system on an very abstract level to prevent major performance loss or they perform a precise hardware simulation and suffer of a slow execution speed. Compared to virtualization using KVM, Simics is 771 times slower and QEMU 33–165 times slower than KVM [17]. Therefore it would be desireable to combine execution speed and the ability for full system analysis.

SimuBoost addresses this need by splitting the simulation in seperate intervals that are simulated simultaneously. Obviously in order to enable a parallel execution of intervals the start system state of each interval has to be known. Therefore SimuBoost utilizes a virtualization to collect system state information for each simulation interval. This approach scales with the run-time of the simulation because the longer the simulation the more intervals are extracted and distributed to parallel simulation nodes and the higher is the speedup by SimuBoost [17].

In order to provide the simulation nodes with system states, herinafter referred to as checkpoints, SimuBoost requires a suited checkpointing mechanism.

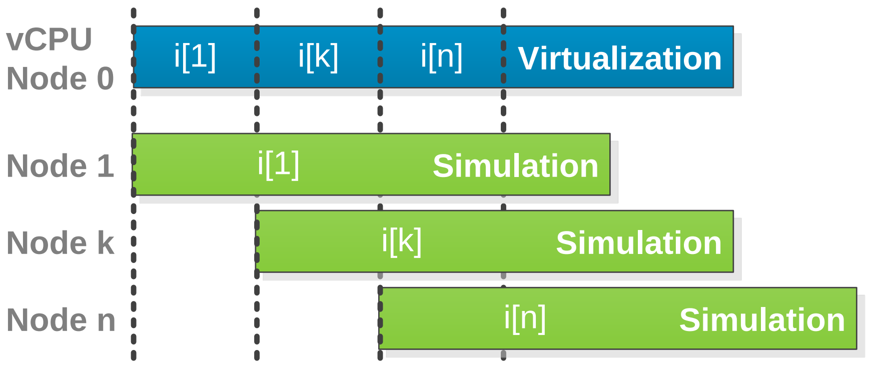

Figure 2.1: Parallel execution of simulations at different points in time on mutliple nodes [17]

2.2.1 Incremental Checkpointing

A checkpoint describes the state of a VM or full system simulation at a point in time. In general this includes main memory, persistent memory, devices and the CPU. In this thesis a checkpoint refers only to an exact main memory image at a point in time.

When a checkpoint is taken, the VM is suspended and restarted after the checkpointing mechanism is finished. The period of time while the VM is suspended is called downtime of the VM. If taking high frequency checkpoints, as done in the application of SimuBoost, the downtime affects the overall runtime of the VM. As a result the checkpointing mechanism affects the interval length of SimuBoost, as a shorter interval results in a higher overhead due to downtimes of the VM (because of more frequent checkpoints) whereas longer intervals influence the runtime of the simulation nodes negatively. Rittinghaus [17] performed calculations resulting in an optimal virtualization execution interval of 2 seconds.

Apparently saving complete main memory, persistent memory and devices states leads to a high data amount. As the amount of data, which has to be processed, correlates directly with the downtime, it has to be reduced to a minimum. Baudis [5] achives this by applying incremental checkpointing in combination with deduplication of main and persistent memory.

In page based incremental checkpointing only modified page frames, hence page frames that have changed since the last checkpoint, are considered by the checkpointing mechanism. As a result the data amount is decreased noticeably, namely about 5–10% [5, 13]. However loading a checkpoint gets more time-consuming given increments are scattered over previous checkpoints and have to get reassembled.

The data amount can be further reduced by deduplicating checkpoint increments. As incremental checkpointing only considers dirty page frames per checkpoint one would expect to see low numbers of duplicates. Still running incremental checkpointing results in high duplication of page frames. Baudis [5, 15] differentiates between two duplication types in his thesis:

- **Intra-checkpoint duplication:** Two or more page frames of the same checkpoint contain the same data.
- **Inter-checkpoint duplication:** Two or more page frames at different checkpoints contain the same data.

He states that intra-checkpoint duplicates are under 10% of dirty page frames while inter-checkpoint duplicates are at 15–55% during a Linux kernel build [5, 15]. The high duplication rate in incremental checkpointing raises the question for semantical properties of duplicate page frames, as detailed information could be used to further reduce data amount and downtime or rather optimize systems to produce less duplicates and use existing memory more efficently. Unfortunately there are no analyses of page frame duplicates in incremental checkpoints.

2.2.2 Characteristics of Main Memory Duplicates

Despite absence of research regarding analysis of duplication in incremental checkpoints, there is research describing characteristics of main memory duplicates in general. They are of interest in the context of virtual machines. If main memory duplicates in simultaneously running VMs can be found, it is possible to deduplicate them and save main memory. Therefore finding of sharing opportunities is one of the main reasons for analysing main memory duplicates.

Baker et al. [4] installed a memory memory tracer based on a Linux kernel module to collect data of interest. It walks through each page of memory, calculates a hash based on the page content and collects additional data regarding the content type of a page frame (e.g. stack, heap, file mapping) and the process using the page frame. They used data resulting from their memory traces to analyze sharing potential of VMs and real world systems and to look into sources of redundant page frames. They found that 67% of page frames could be shared (hence are duplicates) during a Linux kernel build and 45% of duplicate pages come from file mappings and 55% from anonymous memory.

In a study performed by Kloster et al. [13] 93% of page frames during a kernel build were assigned to the page cache, while under 1% were anonymous memory. They used introspection to collect information from the running VM. This approach allows an event driven data recording and keeps the impact on the VM as small as possible.

2.3 Full System Analysis

To analyse a computer system as a whole it is not sufficient to only collect data of selected processes as the analysis misses interactions of operating system components. Therefore it is necessary to collect and analyze data on a system level by executing full system simulators, which are completely observable.

2.3.1 Operating System Introspection

Inspecting the main memory of a virtual machine cannot be done directly, because the inspecting unit acts outside the virtual machines context. It has no understanding of the information that is encoded in the VMs memory. To bridge this semantic gap and gather semantical information from a guest system introspection is used.

In general there are three different methods to bridge the semantic gap. Each with advantages and drawbacks.

Tracing facility in guest context By moving the tracing unit into the context of the guest system there is no more semantic gap that needs to be bridged. Although the approach is easy to use and very straightforward it has a huge effect on the observed system as the tracing unit has to store collected data internally using ressources of the observed system thus falsifying trace data and results. Baker et al. [4] used this approach to analyse redundant page frames (Section 2.2.2).

Gu [11] et al. use an implanted process that executes inside the host under the cover of an existing process to lower the semantic gap of anti-malware software that is placed in the hypervisor of a VM.

Tracing facility outside guest context (with guest modification) In this scenario the actual tracing facility is located outside the context of the inspected VM. Nevertheless the guest operating system that is running in the VM has to actively send data of interest to the hypervisor. The hypervisor receives data and stores it for later analysis. The procedure still has an effect on the guest system as it has to cooperate and send data, but the effect can be minimized dependent on the implementation of the mechanism.

Rittinghaus [15] uses this method to collect data required for a detailed analysis of duplicated memory pages. He records fundamental operating system events to recreate the operating systems memory semantics (including virtual memory of processes) at each point in time. This allows him to make statements about when memory duplication occurs and how long duplicated pages last in memory. Besides that he gets information about how duplicates are distributed amongst memory categories.

Groeninger [10] uses the framework developed out of Rittinghaus diploma thesis to analyze advanced page sharing opportunities. He applies a similar introspection interface to collect data. Using a hidden markov model, he develops heuristics to identify specific workloads that can be used to adjust the scan behavior of a memory scanner to improve page sharing opportunities.

Tracing facility outside guest context (without guest modification) It is possible to move the tracing facility completely out of the guest system and even resign guest modifications that send data to the hypervisor. Julino [12] achives this by using debugging symbols and utilizing the static position of the operating system kernel in the address space of every process to determine and differentiate contexts.

As already mentioned this approach needs no cooperation of the guest system, therefore an unmodified operating system can be executed. On the other hand the use of debugging symbols results in an increased complexity of the implementation, as programm contexts are reassembled in the simulator.

2.3.2 Simutrace

Simutrace [16] is a flexible tracing framework conceived for efficient full system memory tracing. It addresses the limitations of existing tracing frameworks which are only able to track selected processes, therefore missing interaction with the OS, system daemons or (kernel-mode) drivers. To avoid this restrictions, Simutrace captures events at hardware level using functional full system simulation, thus including all user-space programs, operating system events, direct memory accesses and system drivers. As a full system simulation is running an unmodified OS, the tracing has no impact on the observed system and is free of any side effects [16].

Simutrace's tracing capabilities are not limited to memory tracing only. Its design is as general as possible allowing it to adapt to new tracing scenarios and variable data. Simutrace architecture (Figure 2.2) is reflecting this by splitting up into components, representing a classical client server architecture, which allows both local and remote tracing sessions.

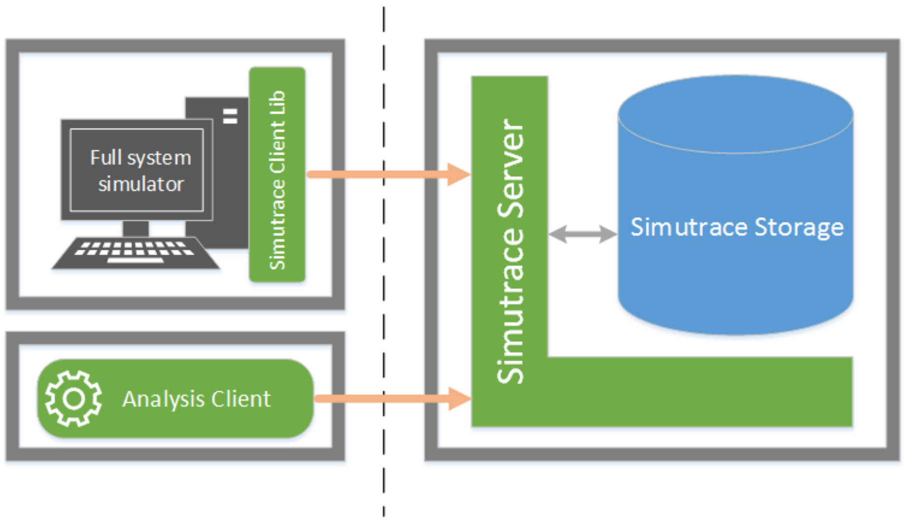

Figure 2.2: Simutrace general architecture

The Simutrace client library extends an existing full system emulator and transfers collected information to the Simutrace server. An analysis client, that uses the same client library as the full system simulator, can connect simutaniously, or after data gathering is finished, to analyze transferred data.

As main component the server manages client connections and presents traces from and to clients. To reduce space consumption it compresses/decompresses traces before they are written to or read from persistent storage.

Traces created by Simutrace consist of sequences of recorded events. Recorded event data is not interpreted by Simutrace, neither places Simutrace any restriction on the data type. As a result it only needs to know the intended data size to offer an event based data access. This is done by registering a new data stream to which data of the specified size is committed. Managing fixed sized streams has one drawback. The approach forbids tracing data whose size is not known in advance such as strings. Therefore Simutrace provides variable sized streams too, which are using references to store variable sized data [16].

There is a patch [21] for the full system simulator QEMU that fully integrates Simutraces tracing capabilities into QEMU by extending its existing tracing backend. Source code concerning trace events, which are defined in a dedicated configuration file, is generated during the QEMU build process and compiled so it is useable immediately.

Chapter 3

Analysis

Baudis [5] measures on average 17.50 % duplicate page frames per checkpoint during a kernel build in a virtual machine with an incremental checkpointing mechanism activated. If it is taken into account that he only considers dirty page frames this number seems very high and raises the question what properties and origin duplicates in incremental checkpoints have. Previous research analyzed page frame duplicates at consecutive points in time to find sharing potential in virtual machines. In the case of incremental checkpointing, deduplicateable page frames concern mostly reappearing contents that already existed in memory at previous checkpoints [5][15].

Therefore existing work can only be taken as a hint at the features of page frames deduplicated incremental checkpointing. Hence it is neccessary to take a deeper look at the memory state during incremental checkpointing. As a result the scenario of Baudis bachelor thesis has to get recreated and extended with mechanisms to inspect the running simulation especially when a checkpoint is made.

3.1 Memory Semantics

An operating system provides an abstraction from underlying hardware enabling user programs to run. One of these abstractions is virtual memory. Each programm gets a fixed range of virtual addresses. If actual memory is needed the operating system maps a page frame to a virtual address. The content of a page frame gains its semantics through the assigned process and the program that is represented by the process. Because the abstraction of the operating system makes it difficult to infer exact semantics, this thesis uses an approximation. Memory that is assigned to a process can be roughly divided into two types of usage.

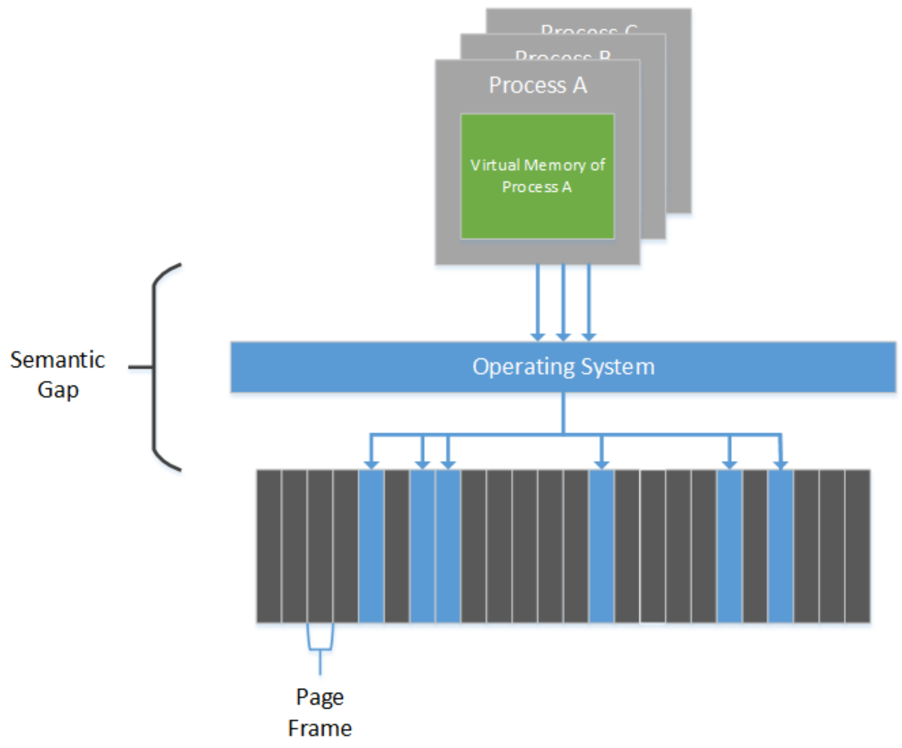

Figure 3.1: Mapping of processes virtual memory to physical memory. Memory semantics arise from opearting systems mapping.

Anonymous Anonymous memory is non-static memory that is used to hold data arising during the execution of a process. It includes processes stacks and the heap. Heap and stack can grow dynamically depending on the memory demand of the process.

File backed Memory that contains file mappings from disk is called file backed. File backed memory includes mappings of executeables like libraries or a programs binary code and files that are currently read and/or written by any system component.

Therefore in this thesis, the semantics of a page frame consist of the process that requested a physical to virtual memory mapping by the OS and the type of usage, which is determined by the OS (Figure 3.1). Accordingly without the context of the operating system (and the program/process the memory is assigned to), memory semantics of page frames are lost or rather hard to recover.

This means in order to characterize duplicate page frames of incremental checkpoints it is necessary to extract information from the operating system that can be used to distinguish between anonymous and file backed memory and to determine the process a page frame is assigned to.

Modern operating systems implement page frame protection mechanisms on top of the physical to virtual memory mapping facilities, to prevent unauthorized manipulation. In fact various access rights for page frames are defined by the OS. A page frame can be readable, writeable and/or executable. Depending on the OS there can be more access rights. The access rights defined by the operating system can be used to distinguish between anonymous and file backed memory.

In general page frames of a file backed memory mapping that targets binary code are executable and read-only to prevent malfunction by overriding the page frame accidentally. In case of a common file the memory mapping marks the page frame readable and/or writeable but not executable.

If a physical to virtual memory mapping is not file backed, the targeted page frame is identified as anonymous memory. Anonymous memory is readable and or writeable though not executable.

Beside these process dependent memory types there is information in the absence of a physical to virtual memory mapping too. Therefore the dissolving of a memory mapping by the OS, which results in a free, reuseable page frame carries usage information as well. Furthermore the OS maintains file caches to speed up file access by holding file references in unused page frames. Hence if a process wants to access a file that is already located in a file cache, it gets mapped in to the processes virtual address space. As the page cache in general is process independent it is a second source of information about the usage of page frames, which does not rely on the presence of a memory mapping.

In table 3.1 the derived page frame usage types, hereinafter called allocation types, and their attributes are summarized. The attributes can be used to identify data needed to determine a page frames allocation type.

Allocation type	**Attributes**
anonymous memory	readable and writeable, no file mapping
free memory	no specific attributes, freed by operating system
file backed	only readable, file mapping
operating system cache	operating system dependend

Table 3.1: Allocation types and their attributes derived from operating systems virtual memory abstraction.

3.2 Data Acquisition

Basis for page frame usage is information derived from the different memory mapping and operating system mechanisms (Figure 3.1). By evaluating the access rights and testing if there is a file mapping established, it is possible to differantiate between anonymous memory and file backed memory. As a result corresponding data has to get collected from the guest system by tracing the operating systems page frame allocation facility. By using data extracted from this mechanism it is possible to define allocation types that determine memory semantics.

As there is no possibility to derive if a page frame got freed from the attributes itself, the freeing mechanism of the operating system has to get identified and traced as well. The same applies for operation system caches.

To this point the data acquisition approach does only record page frame allocation therefore it is not possible to decide which process triggered the memory allocation. Consequently scheduling operations of the operating system have to get traced too.

Operating system facility	**Data of interest**
process creation	PID, executable filename
process scheduling	PID
allocate page frame	64 bit page frame address, allocation type
free page frame	64 bit page frame address
add page frame to cache	64 bit page frame address, cache identifier
remove page frame from cache	64 bit page frame address

checkpointing event	**data of interest**
insertion of a page frame	64 bit page frame address, checkpoint, data hash
deduplication of a page frame	64 bit page frame address, checkpoint, data hash

Table 3.2: Data needed for recreation of the memory semantics and correlation of this data with the checkpointing information

To determine the actual running process, process information like process identifier (PID) and an executable filename have to be known. They can be intercepted during process creation. The running process can be identified by trapping the OS scheduler and sending the PID of the running process to the hypervisor. As a process had to be created before it can run, the received PID can be correlated with an executable filename during analysis. In table 3.2 the key facilities of the operating system, that need to be traced to recreate page frame usage at runtime, are summarized.

Even though mentioned data is sufficient to analyze page frame usage and approximate memory semantics there is no connection to the checkpointing mechanism hence it is not known which page frames changed since the last checkpoint and which of the page frames are duplicates. Insertion of new page frames and deduplication of already known page frames is completed by the checkpoint storage. Therefore these specific data has to get extracted every time a checkpoint occurs. Additionaly it is possibly of interest which content a dirty or rather deduplicated page frame has. As the overhead of saving the whole content of a page frame is too big, it is sufficient to extract the calculated data hash, which is used to find duplicates.

Table 3.2 states the data required to form a connection between checkpointing and OS data. The correlation of checkpointing and OS data also needs a suited timing mechanism to make sure the memory semantics get correlated with the correct checkpoint.

3.2.1 Timing Facility

As a checkpoint occurs at a dedicated point in time a facility is needed that adds timestamps to collected operating system and checkpointing meta data. Without these timestamps there is no way to guarantee a correct temporal order of events. Additionaly a timestamp is needed to correlate checkpointing metadata with operating system data. To do so all operating system events till the first checkpointing event of the examined checkpoint have to get passed through. Afterwards it is possible to analyze semantics of page frames delivered by the checkpointing events.

Chapter 4

Design

To analyze page frame duplicates of incremental checkpoints, the semantic background of page frames needs to be known. Unfortunately the data cannot be extracted straightforward because of the semantic gap between the host and the guest system. Therefore dedicated operating system facilities are traced to recreate approximated memory semantics, as an exact reconstruction of the memory semantics is too complex. The approximation describes the usage of page frames by processes and the opearting system by defining allocation types.

4.1 Closing the Semantic Gap

To recreate memory semantics important operating system data has to be actively transferred from the guest system to the host system.

There are two introspection methods to achive a data transfer from the guest to the host system. The first approach makes use of debugging symbols and connecting a debugger to the virtual machine. A big advantage of this approach is that an unmodified operating system can be used, which minimizes the effect on the observed system. However, the implementation overhead is very high as a simple realization lacks in execution speed whereas fixing the execution speed issue noticeably increases implementation complexity [12].

The second approach relies on modifications of the guest operating system at relevant source code locations to collect data and send it so the hypervisor, which processes and stores the data. The procedure of sending data is realized by calling a special instruction that invokes the host. Apparently this approach has an effect on the guest operating system as it is modified to collect and send data. Nevertheless it is a good trade-off between the influence on the guest system and a moderate implementation overhead.

The actual introspection process can be split up into two parts as shown in

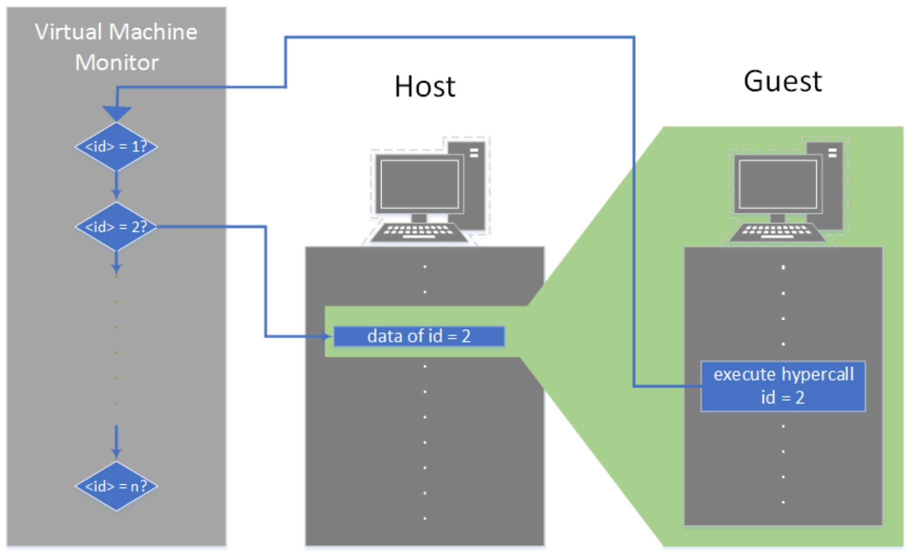

Figure 4.1: Hypervisor invocation schema

Figure 4.1. The first part is to trap the operating system when an event of interest occurs and send the virtual address of the data, which is collected in the guest operating system, to the hypervisor. The second part includes address translation and data fetching from the hypervisor.

As the guest is a self-contained system, virtual memory addresses received in the hypervisor are invalid in the host system. To read the actual data, virtual guest addresses have to get translated into virtual host addresses. When address translation and data access are finished and the data was send to the storage server the control flow returns to execute the guest simulation. While bridging the semantic gap the impact on the guest system should be kept as small as possible to prevent maloperation and the apperance of wrong data.

As indicated in Figure 4.1 a finite set of recordable events has to get predefined and supplied with an identifier to determine how many bytes of data are read from the guests memory and how it is structured. To guarantee a correct temporal order of events a timestamp is added to every event. It is sufficient if the used timing mechanism gurantees a monotonic increasing order of events.

4.2 Checkpoint Metadata

The events provided by the OS introspection are missing information about page frames that got newly inserted in the incremental checkpointing or deduplicated by it. To collect checkpointing metadata the deduplication mechanism has to get instrumentalized.

The deduplication mechanism, which is embedded in the incremental checkpointing, calculates a data hash of every page frame content that comes with a checkpoint. The data hash is compared with data hashes of already inserted page frame contents. If two data hashes match, which means a page frame is deduplicated not newly inserted, an deduplication event is triggered and stored with the OS introspection events. If there is no match with an already stored data hash, the page frame is newly inserted and a insertion event is triggered and stored with the OS introspection events as well.

Event hough the instrumentation of the deduplication mechanism provides checkpointing events, which identify deduplicated and newly inserted page frames per checkpoint, there is no reference point, which can be used to correlate checkpointing events and OS introspection events. In Section 4.1 it is stated that OS introspection events get timestamps from a timing mechanism that guarantees a monotonic increasing order of events. This timestamp is a suited reference point for correlation of checkpointing events and introspection events, therefore checkpointing events need a timestamp as well. To ease correlation every checkpointing event that belongs to a dedicated checkpoint gets an identical timestamp. As a result a sequence of alternating OS introspection and checkpointing events is produced. All OS introspection events that occured before the first of a sequence of checkpointing events, determine the system state of the checkpoint described by the current sequence of checkpointing events (Figure 4.2).

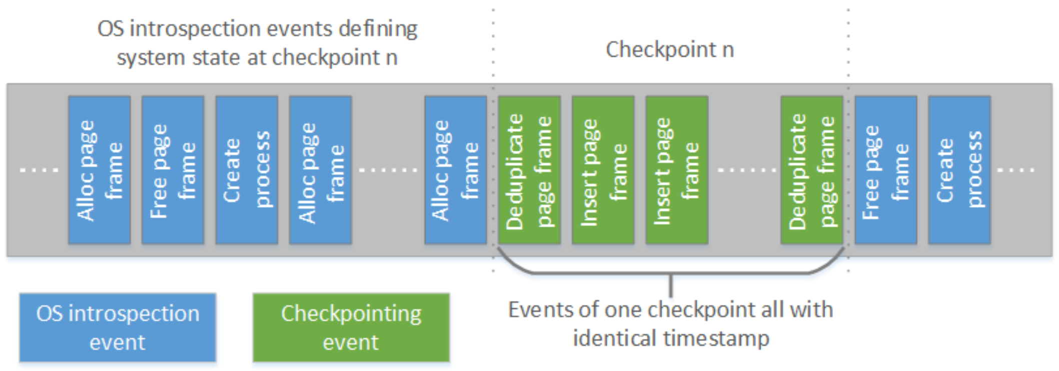

Figure 4.2: Sequence of alternating OS introspection and checkpointing events.

4.3 Intended Data Analysis

Event based introspection, combined with event based checkpointing metadata leads to a set of sequences of temporal ordered events (Figure 4.2). The intended analysis splits into two steps. First it reassembles the allocation type and the process context in which an allocation took place for every page frame at every checkpoint by iterating through the event sequence. This approximates the original memory semantics, which existed at runtime. Afterwards the resulting data model representing the abstract system state at every checkpoint is analyzed.

There are three different kinds of events.

Scheduling events include creation and scheduling of a process. When a new process creation event occurs, the executable filename and the process ID are getting saved and used if a scheduling event occurs that contains the same PID. Process schedule events change the current active process to allow assigning of arising allocation events to a process.

Allocation events include allocations, freeing of page frames and adding/removing of page frames to/from operating system internal caches. Each allocation event determines the current semantics of a page frame. When a allocation event occurs the allocation type of the given page frame is changed to the new allocation type.

Checkpointing events include events that mark an inserted or deduplicated page frame at one checkpoint. All checkpointing events of a checkpoint contain the same cyclecount as the virtual machine pauses when a checkpoint occurs. When a checkpointing event occurs the referenced page frame and its data hash are getting saved with the checkpoint they got inserted or deduplicated in.

While iterating through the event sequence, the inteded action of the current event is executed by updating a data model, which represents the abstract system state per page frame. The used data model splits into submodels, representing the allocation type of each page frame, the known process IDs and corresponding executable filenames and a model managing checkpoints and associated insertion and deduplication events. As checkpointing events always appear bundled in the event sequence (Section 4.2), they indicate the occurence of a checkpoint. If a checkpoint is noticed in the event sequence the current state of the data model is related to the current checkpoint and its events, so it can be analyzed in a second step.

When the reassembling step is finished, the analysis step creates statistics of every checkpoint by using the data-model-checkpoint-relation created in the first step. The intended analysis counts allocation types of inserted and deduplicated page frames per checkpoint and calculates percentages to describe their distribution. To find distinctive features in the allocation behavior of specific processes their allocation types over the course of the checkpoints are counted as well. In order to find correlations between specific page frame contents and allocation types, data hashes of the checkpointing events are analyzed by counting their occurence in combination with the allocation types.

Because of the event based introspection approach and the resulting data model at each checkpoint it is very easy to extend the analysis and adapt it to arising requirements.

Chapter 5

Implementation

The checkpointing mechanism used in this thesis was implemented by Eicher [8]. It is a modified version of the checkpointing mechanism that was integrated into QEMU/KVM by Baudis [5]. Eichers implementation replaces the storage backend used by Baudis with a faster storage mechanism by using Simutrace and asynchronous data processing.

In this work KVMs virtualization and the speedup associated with it is renounced, as the checkpointing mechanism is embeded in QEMUs VMM and abstracted from KVM. It has the advantage that QEMUs existing tracing infrastructure and functionality provided by the earlier mentioned Simutrace patch [21] can be used. This includes an cyclecount, which can be used as time source for events and is only available in the TCG mode of QEMU. Hereinafter, QEMU with the applied Simutrace patch is referred to as QSimu.

The use of QSimu to run the virtual machine allows the use of Simutrace to store introspection and checkpointing events. QSimu is based on version 2.2.1 of QEMU while the checkpointing mechanism of Eicher was implemented in QEMU version 1.5.50. So the checkpoiting mechanism got ported to QSimu which made it neccessary to disable disk checkpointing because of incompability, which is not important for this thesis anyway.

The Simutrace server used in this thesis was extended by Eicher [8] to store and deduplicate received incremental checkpoints.

The investigated guest system is a Linux system with the kernel 4.1.0 as its source code is available and it is documented well. It runs on an emulated x86-64 CPU. The first problem that has to get addressed is bridging the semantic gap, hence sending operating system events from the host to the guest system.

5.1 Operating System Introspection

The guest running in QSimu is a semantically closed system. QSimu does not provide an earlier mentioned magic instruction that allows to make a call from the guest system into the host system. As a result a custom mechanism has to be used.

The x86 CPU architecture provides two instructions, `rdmsr` and `wrmsr`, for reading and writing model specific registers (MSR). MSRs can be accessed by defining a 32 bit index that has to be loaded into the `ECX` register before the MSR instruction is called. The MSR instructions define an unused register address range from `0x40000000` to `0x400000FF` that has not been used in any past and will not be used in any future CPU implementation [1].

If a MSR instruction with an invalid register address is executed on a real machine the CPU throws an exception. In QSimu a dedicated helper function handles the `rdmsr` and `wrmsr` instructions. This fact can be used to trap the simulator into receiving semantic information from the guest system.

In this thesis the `rdmsr` instruction is used to hook the simulator up to receive data from the VM, as just reading a value from a MSR does not influence the system as much as writing a MSR. The call of the VM into the simulator is referred to as executing a hypercall.

5.1.1 Hypercall Interface

To ease the use of hypercalls a function that executes the hypercall is declared. It takes two arguments, an 8 bit event identifier (EID) to differantiate between OS events and a pointer to the data structure that is send.

As the size of the unused MSR address range is exactly 8 bit, the EID is transferred by adding it to the lower limit of the unused MSR address range. Therefore there is no usage of an additional register to transfer the EID, hence less influence on the running system and no guest memory access from the simulator to retrieve the EID.

The event data getting transferred to the host is organized in a packed C structure. A pointer to the structure is passed to the hypercall function that stores it in the 64 bit `RDX` register. By packing the structure containing the collected data, hereinafter referred to as *hypercall structure* (HS), we prevent the compiler from inserting paddings that could break data access from the simulator. In the simulator, the EID is interpreted and determines the HS that is read from the guest memory. Hence the simulator, QSimu, has to have knowledge about the data hypercall structures used for data transmission in the Linux kernel to gurantee a correct access and interpretation of the data.

When a hypercall is triggered in the guest operating system, the `rdmsr` instruction is handled in QSimu by the dedicated helper function. It checks if the value in the `ECX` register – the MSR address – is equal to the start of the reserved MSR address range and extracts the EID. Afterwards it calls into a function translating the guest pointer stored in `RDX` into a host pointer that can be used to access the data defined by the EID.

The address translation function uses QEMUs internal *SoftMMU* to get corresponding virtual host addresses from virtual guest addresses. In general, when virtual guest addresses are translated into virtual host addresses, the translation is accomplished in two steps. A translation from a guest virtual address to a guest physical address and afterwards an translation to a host virtual address is executed.

The SoftMMU of QEMU is basically a translation lookaside buffer (TLB) that caches the offset of a virtual guest address to a virtual host address. This implementation disregards the fact, that a software implementation of a hardware TLB would usualy only store the offset between virtual guest and physical guest address.

If there is an entry for a virtual guest address in the SoftMMUs TLB, the offset is used to receive the virtual host address directly, otherwise QEMUs `tlb_fill(...)` function is called to process the virtual guest address and store the calculated offset in the TLB. This process has no impact on the VMs system state because it has no knowledge of QEMUs SoftMMU.

The resulting virtual host address is passed to a *hypercall handler function*, casting it to a pointer to the corresponding HS and storing the data via Simutrace.

This procedure affects the guest system state at a minimum, as all used registers are saved before a hypercall is executed and restored when the control flow returns from the host.

Furthermore collected data is not stored on the guest system heap, as this solution involves calls of `kmalloc`, the linux kernels memory allocation function, which would influence the guest system state, in worst case triggering a page allocation falsifying results. The collected data is stored on the current process' stack instead, as it minimizes the effect on the guest system memory state.

5.1.2 Operating System Events

The hypercall interface is used at particular locations in the Linux source code where data of interest is available and can be intercepted.

In chapter 3.2 traced operating system events were stated. They are general descriptions for specific, operating system dependend implementations which have to be located in the Linux source code. Figure 5.1 gives an overview of installed hypercalls and their source code locations.

The principal source for relevant semantic information are page frame allocations. As there is no central location in the Linux source code where the allocation

Event	**Intercepted function**	**Source file**
process creation	do_fork(...)	kernel/fork.c
process scheduling	context_switch(...)	kernel/sched/core.c
		arch/x86/include/asm/...
page frame allocation	native_set_pte_at(...)	pgtable.h
	native_set_pmd_at(...)	pgtable.h
	native_set_pte(...)	pgtable_64.h
	native_set_pmd(...)	pgtable_64.h
adding page frame to page cache	add_to_page_cache_locked(...)	mm/filemap.c
removing page frame from page cache	delete_from_page_cache(...)	mm/filemap.c
add page frame to slab	allocate_slab(...)	mm/slub.c
removing page frame from slab	free_slab(...)	mm/slub.c

Table 5.1: Source code locations of hypercalls installed in the Linux kernel

of page frames is dealt with, the hypercalls regarding general allocations are installed in the facility that establishes virtual page to page frame mappings. The allocation type of a mapping is derived by evaluating the Linux kernels memory descriptor structure (`mm_struct`).

It contains all information about a processes address space by holding a list of memory regions. Linux describes a memory region by a structure of the type `vm_area_struct`. It defines a contiguous linear address interval that never overlaps with other memory regions of the same process. It is possible to retrieve the corresponding `vm_area_struct` of a page frame by calling the function `find_vma(...)` and passing it the page frame address and its memory descriptor. Each memory region defines its own access rights by setting corresponding flags. If a memory region is file backed, it contains a pointer to the related file object. Therefore by checking if the file object pointer is `NULL` and evaluating the access rights it is possible to differentiate between an anonymous memory region and a file backed memory region.

Besides page frames that are allocated by processes there are page frames used by operating system caches. The Linux kernels maintains two caches, the page cache and a facility called slab allocator.

The page cache of the Linux kernel is a common disk cache. It is used to speed up file accesses by keeping frequently used data in main memory. It is invoked when a process wants to read from or write to a file. If a file is not already in the page cache, the kernel adds new page frames to the page cache to satisfy the request [7]. The slab allocator allocates page frames to hold often used kernel data structures. A slab only holds one specific data type. If the kernel requests such a data type, instead of allocating and initializing a new memory area, the kernel reuses an existing one of the slab [7]. Both facilities contain hypercalls to gain information which page frames are assigned to them.

The set of introspection events compromises many different kernel source code locations that have to be modified. The modification in detail consists of collecting needed data, creating the HS on the stack and executing the hypercall with the EID and the HS pointer as arguments. For every operating system event that is traced in the guest operating system, QSimu needs a handler function that accesses the data and stores it in Simutrace. The procedure of creating a HS, executing a hypercall and processing it in QSimu is in general analogical. When a HS is created the fields of a C structure are initialized with predefined, event dependent values. The following hypercall execution is similar for all events and the data access in QSimu only differs in the HS structure that is determined by the EID. Finally the storage of received events is accomplished by calling functions, which were generated from a configuration file by QEMUs tracing backend.

Because of mentioned similarities in the implementation of hypercalls and data processing in QSimu, dynamic code generation is used.

5.2 Full Code Generation

As discussed in chapter 5 QSimu extends QEMUs existing tracing infrastructure to provide a storage interface for Simutrace. QEMUs tracing infrastructure is generated when QEMU is built. It reads its tracing events from a configuration file that defines a tracing function name, function arguments and a standart text output of the arguments and generates a callable C function. This function, in the following referred to as tracing function, takes declared arguments and stores them bundled into a dedicated Simutrace stream. Addtionaly it automatically adds the cyclecount timestamp to the bundled data.

By inserting the corresponding generated tracing function into the event dependent hypercall handler function in QSimu, mentioned in section 5.1.1, there it is possible to generate data processing procedures for every event defined in QSimus trace event configuration file.

As noticed in section 5.1.2, hypercalls from the guest operating system to QSimu only differ in the HS used for data transmission and their initialization. Information concerning the needed fields of a HS can be extracted from the trace event configuration file as well. If the needed fields of a HS are known it is possible to generate a function for the use in the kernel, which initializes the HS on the stack. As the hypercall itself is already encapsulated in a function it can be added to the initialization function, taking a pointer to the recently created HS and the corresponding EID. As a result the source code responsible for creating a HS and invoking the hypercall with the pointer and EID as argument is fully dynamically generated.

```
void send_<EVENT_NAME>_struct(<ARGUMENT_1> field_1, ...,
                        <ARGUMENT_n> field_n) {

        <EVENT_NAME>_t <EVENT_NAME>;y
        <EVENT_NAME>.field_1 = field_1;
        ...
        <EVENT_NAME>.field_2 = field_n;

        execute_hypercall(HYPERCALL_ID_<EVENT_NAME>,
                        &<EVENT_NAME>);
}
```

Listing 5.1: Structure of a generated function that initializes a HS and executes a hypercall

Listing 5.1 shows the general structure of a function that is generated and callable at appropriate locations in the Linux kernel. Because the counterpart located in QSimu is completely generated as well this leads to a full code generation for operating system introspection events and their storage via the Simutrace framework.

The full code generation provides an easy to use introspection tool for QSimu running a an accordingly modified kernel. Adding a new trace event to QSimu and the kernel is done by inserting the event definition into the trace configuration file and running the build script as shown in Figure 5.1.

The build script first calls the introspection code generator that is written in Java. It reads the trace configuration file and generates specified Linux and QSimu source code and makefiles. Supporting source files are copied to the paths declared in a generator configuration file. Afterwards, the build script starts building

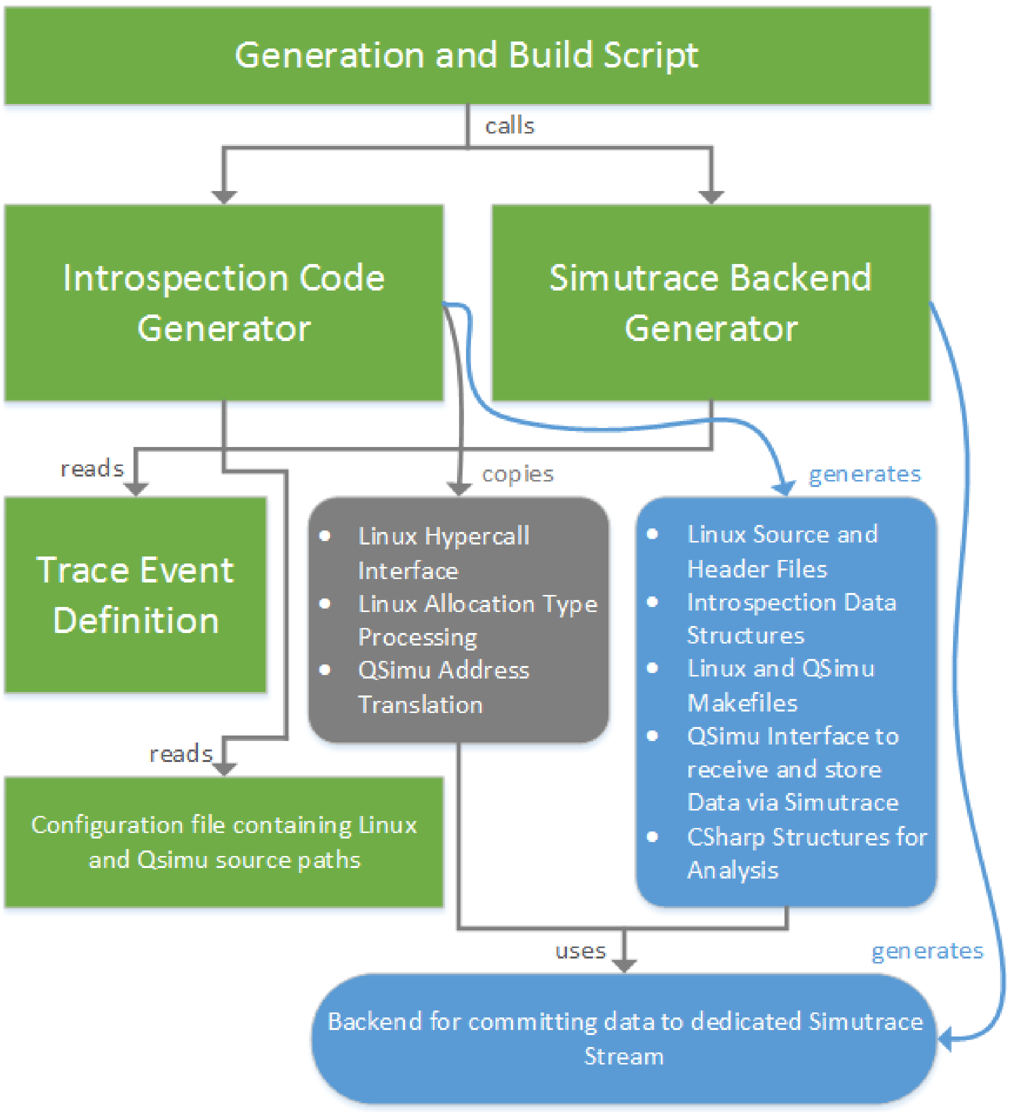
Generation and Build Script
calls
Introspection Code Generator
Simutrace Backend Generator
reads
copies
generates
Trace Event Definition
Linux Hypercall Interface
Linux Allocation Type Processing
QSimu Address Translation
Linux Source and Header Files
Introspection Data Structures
Linux and QSimu Makefiles
QSimu Interface to receive and store Data via Simutrace
CSharp Structures for Analysis
reads
Configuration file containing Linux and Qsimu source paths
uses
generates
Backend for committing data to dedicated Simutrace Stream

Figure 5.1: Schema of the code generation mechanism. Blue represents generated code.

QSimu which results in a call to the QSimu intern tracing code generator that creates the Simutrace backend from the trace event configuration file. Finally QSimu is compiled and linked with the new introspection interface. In order to use the newly generated events, the created Linux source files are copied to the operating system[1] that is running in the simulator and the kernel is rebuilt.

[1]In this thesis generated kernel source code is pushed to a version control system and pulled inside the simulation

Furthermore, it is possible to extend the code generation to generate code for an easier analysis. At this point it just generates additional HS in C# for later use in an analysis tool that interfaces with Simutrace as well.

5.3 Checkpointing Metadata

Simutrace provides a flexible storage backend, which is adaptable to individual requirements. In the Simutrace version that is used in this thesis two storage backends are available. Checkpointing data is handled by a storage backend that was implemented by Eicher [8]. It includes a deduplication mechanism based on a hash map, using a data hash over the page frames content as a key and the content itself as value. Trace events are stored using the standard storage format of Simutrace.

Simutrace is a client server architecture and QSimu just sends modified page frames to the server without having knowledge of page frames that are deduplicated by the storage backend. Therefore it is required to instrument the deduplication facility to gather checkpointing metadata. Checkpointing metadata includes information about inserted page frames and deduplicated page frame. It is needed to allow an analysis of modified page frames and duplicates in the incremental checkpointing mechanism.

In chapter 4.2 it was derived that the cyclecount from QSimu is needed to enable a correlation between checkpointing metadata and introspection data. As a result this data has to be activily transferred from QSimu to the storage server. To do so two new API calls are introduced in Simutrace:

StIntrospectionSetStore `StIntrospectionSetStore` just is invoked once when the checkpointing mechanism is started. It transfers the identifier of the introspection storage to the Simutrace server, so checkpointing metadata and introspection data are saved in the same storage.

StCheckpointCreateAtCyclecount `StCheckpointCreateAtCyclecount` matches an existing API call but takes the cyclecount at which the checkpoint creation occurs as additional argument. Beside creating a checkpoint, it invokes an Simutrace intern function that executes a predefined remote procedure call (RPC) transfering the data from the client to the Simutrace server.

In the Simutrace server application a data collector class receives cyclecount updates and manages the connection to the introspection storage. The class is realized as singleton, to encapsulate the intercepting code and to seperate it cleary from the Simutrace server. As the class is implemented as singleton, it does not create new dependencies because of passing needed class references.

When a checkpoint occures, the deduplication mechanism iterates through modified page frames received from QSimu. The deduplication mechanism checks whether a page frame is inserted or deduplicated and invokes the data collector. The data collector accordingly creates a new insertion or deduplication event, which consists of the page frame address, the data hash and the actual cyclecount. Afterwards, the created event is stored in a dedicated stream in the introspection storage.

This results in an storage containing combined data of introspection and checkpointing. As all events that got persisted hold a timestamp it is possible to correlate them in an analysis tool and recreate an approximation of the memory semantics at each checkpoint.

5.4 Analysis Tool

For analysis of the collected data a tool is used that is based on a C# binding of the Simutrace API. Simutrace stores collected data in dedicated streams, namely one stream per event type. For analysis purposes it is of avail if data is merged into one temporal ordered stream, because it facilitates the reassembly of memory semantics. Merging available streams is achived by a Simutrace API call.

It builds a new stream ordered by cyclecounts whose entries are of the type `MultiplexerEntry`, which consist of a stream identifier, a pointer to the actual data and the cyclecount itself. Consequently all events in the multiplexed stream that were recorded by the OS introspection and the deduplication facility of the checkpointing mechanism are in the same temporal order as they occured in the running simulation. With this in mind it is feasible to iterate through the stream's events and reenact their effect on the semantics of page frames.

In order to reenact OS introspection events there has to be a data structure maintaining semantical information for each page frame. For this purpose the analysis tool uses instances of a class called `PageFrameInformation`. It holds, as the name suggests, semantical information of the page frames including the page frame address, the allocation type and a structure, named `ProcessInfo`, representing the assigned process. To identify deduplicated page frames, the `PageFrameInformation` class references an instance of the `CachedDataInfo` class.

It encapsulates the data hash and the checkpoint at which the data hash occured for the first time. An additional hash map takes checkpoints as keys and lists of referencing page frame addresses as values allowing the tracking of reoccurences of data hashes. Table 5.2 summarizes introduced structures and classes:

Struct/Class	Attributes
`PageFrameInformation`	page frame address, allocation type, `CachedDataInfo` reference, `ProcessInfo` instance
`ProcessInfo`	process identifier (PID), executable filename
`CachedDataInfo`	data hash, checkpoint of first occurence, hash map with checkpoints as keys and lists of referencing page frame addresses as values

Table 5.2: Introduced structs/classes and their attributes

A hash map using page frame addresses as keys and `PageFrameInformation` instances as values is maintained to keep track of changes while iterating through the multiplexed stream. Another hash map is maintained to correlate PIDs and `ProcessInformation` instances and a third one to correlate data hashes and `CachedDataInfo` instances. In both cases the PIDs/data hashes function as keys and the structure/class instances serve as values.

Given these data structures one is able to iterate through the multiplexed stream and update the page frame information hash map according to read OS introspection events. The multiplexed stream provides its data by using a specific data type (`MultiplexerEntry`) as mentioned earlier. As a result the identifier attribute of the multiplexed stream is interpreted to determine the type cast that is applied on the actual data pointer, before it is possible to access and reenact an event.

The analysis tool differantiates between three kinds of events as stated in 4.3, scheduling events, allocation events and checkpointing events:

Scheduling events Scheduling events contain process creation and process scheduling, updating the current running process and the process info hash map. When a new process creation event is read from the stream, the analysis tool instantiates a new `ProcessInfo` structure and inserts it into the process info map. When a process scheduling event arises the current running PID gets updated. It is referenced when the allocation event occurs, to correlate allocation event and the current running process.

Allocation events Allocation events involve all events that change the memory semantics. Therefore they include page frame allocation/freeing by a process, adding/removing of a page frame to/from the page cache and adding/removing a page frame to/from a slab cache. When an allocation event is read from the multiplexed stream the exact event is determined and the allocation type and process that triggered the allocation is updated in the page frame information map. On the whole this results in an semantical memory image that is analyzed later on.

Checkpointing events Checkpointing events include insertion and deduplication of page frames. When a checkpointing event occurs the corresponding `CachedDataInfo` instance is retrieved from the responsible hash map or a new one is created. A reference to the page frame where the data hash appeared is inserted and the correlated page frame information gets a reference to the `CachedDataInfo` instance as well.

Checkpointing events of one checkpoint are always tagged with an identical cyclecount. Hence it is possible to identify checkpoints in the multiplexed event stream and copy the state of the page frame information map at every checkpoint for analysis. As a result there is a hash map with `PageFrameInformation` instances for every page frame address at every checkpoint that carries all information needed to achive a detailed analysis of features of modified and deduplicated page frames in incremental checkpointing.

Chapter 6

Evaluation

In this chapter the proposed approach is evaluated and discussed. As this thesis deals with semantical features of modified and duplicated page frames occuring in incremental checkpointing the thesis on hand targets to clarify which semantic types of page frames are involved in incremental checkpoints and how they are distributed. Furthermore, page frame hashes of deduplicated page frames are analyzed to discover eventual conspicuities in their occurence and determine whether duplicates arise from a wide range of page frame contents or a small group of recurring contents.

6.1 Methodology

This evaluation heavily depends on the checkpointing mechanism implemented in the work of Eicher [8], it is assumed that it works correctly. Furthermore, measurements regarding incremental checkpoints of the work of Baudis [5] are used as reference points. To provide a reliable dataset for the evaluation, simulations with different workloads are ran three times respectively and tracing data is collected. The checkpointing mechanism is started when the execution of the workload in the simulation starts and stopped when the 50th checkpoint was created, so collected data is comparable with data of Baudis work, as he took 50 checkpoints in his evaluation as well. QSimu is started in snapshot mode to prevent persistent modifications of the running system and assure an exact same disc image at each run.

An integrated facility is coupled with the checkpointing mechanism of Eicher. It saves statistics of page frames into a text file. The statistics include dirty and deduplicated page frame counts per checkpoint, which can be used to verify the correctness of dirty and deduplicated page frame counts intercepted by the data collector placed in the Simutrace checkpointing storage backend.

Suited workloads for the evaluation are given by the work of Baudis, as the statistics created in his work serve as an important indication for comparable data. By using a kernel build (version 4.1.0-rc2) and the SPEC CPU 2006 401.bzip2 benchmark two fundamentally different workloads are executed. A kernel build stresses the CPU, memory and I/O devices as well. Whereas the 401.bzip2 benchmark compresses and decompresses six files, varying in size and file type, using three different compression levels which leads to I/O activity when reading files into memory and stresses the CPU during compression and decompression. As the workloads of the 401.bzip2 benchmark does not last for the duration of 50 checkpoints, it will run repeatedly and every 10th run the page cache is flushed by a script, so it has to get refilled. This procedure generates a high number of duplicates, as the 401.bzip2 benchmark processes the same files three times and is executed repeatedly. The flushing of the page cache should be visible in the trace data by a decreased number of duplicates.

Tracing data collected during startup, workloads and shutdown is processed by the analysis tool developed in this work. Since the tool reassembles per page frame semantics at each checkpoint from the traced operating system and checkpointing events, it is possible to run multiple analysis on each single trace. The analysis tool processes the traces three times generating the following different statistics.

General statistics, that are based on page frame property counts, are created. They describe dirty and deduplicated page frames per checkpoint. Furthermore, the per checkpoint distribution of allocation types for all dirty page frames and for deduplicated page frames is created.

Allocation types used to categorize page frames are derived from the semantic information introduced in chapter 3.1. Therefore a page frame can be categorized as:

- **Anonymous memory:** Memory that is assigned to a process and not file backed. It stores process private data, which may change during execution. Stacks and heaps are accounted to anonymous memory.
- **File backed:** Memory that is assigned to a process and holds a mapping of a file. It includes libraries, files that are processed by a program or the program binary code itself.
- **Page cache**: Page frames that are assigned to the page cache are not are not mapped by a proces. It correlates highly with file backed page frames.
- **Slab:** Includes all page frames that are assigned to a slab allocator, hence are used for kernel structures.
- **Free:** All page frames that are freed by the buddy allocator are summarized as free page frames.

- **In transition:** When a page frame is removed from the slab or the page cache, it is categorized as in transition until the buddy allocator handles and eventualy frees it.

To supplement these data and to gain information about page content occurence frequency, the data hashes of page frame contents are counted per allocation type and checkpoint. Finally, to retrieve data describing the process usage of allocation types in memory, a third analysis approach counts allocation types per process. All used analysis approach are summarized in table 6.1. By evaluating this data it is possible to make statements about characteristics of duplicate page frames in incremental checkpointing.

Approach	**Resulting data**
general allocation type analysis per checkpoint	distribution of allocation types per checkpoint, general statistics concerning modified and deduplicated page frames
data hash based analysis	data hash counts per allocation type including all checkpoints
process belonging	allocation type distributions of processes

Table 6.1: Approaches used to analyze memory semantics provided by the analysis tool.

Unfortunately, there are page frames, that are not captured by any installed hypercalls or cannot be assigned an allocation type. They get a special allocation types, which indicate the absence of semantic information:

- **Not determined:** The allocation types anonymous and file backed, are derived from the Linux kernels memory descriptor when establishing a virtual to physical memory mapping. However it is possible that there is no memory descriptor structure passed to the establishing function, hence the used approach cannot derive an allocation type.

- **Unknown:** There are page frames that are modified and or deduplicated with no collected semantic information at all. In that case page frames are categorized as unknown page frames.

6.1.1 Checkpointing Interval Adaption

The benchmarks are executed inside the simulation with active checkpointing to trigger duplication and collect semantical information for analysis of duplicates and modified page frames. Baudis has measured on average 25000 dirty page frames per checkpoint, taking 50 checkpoints during a kernel build with a checkpointing interval of 2000ms and hardware virtualization with KVM [5]. As this thesis concentrates on observability and therefore emulates the system, it results in a slowdown and less dirty page frames per checkpoint leading to less duplicates per checkpoint and no proportionate data.

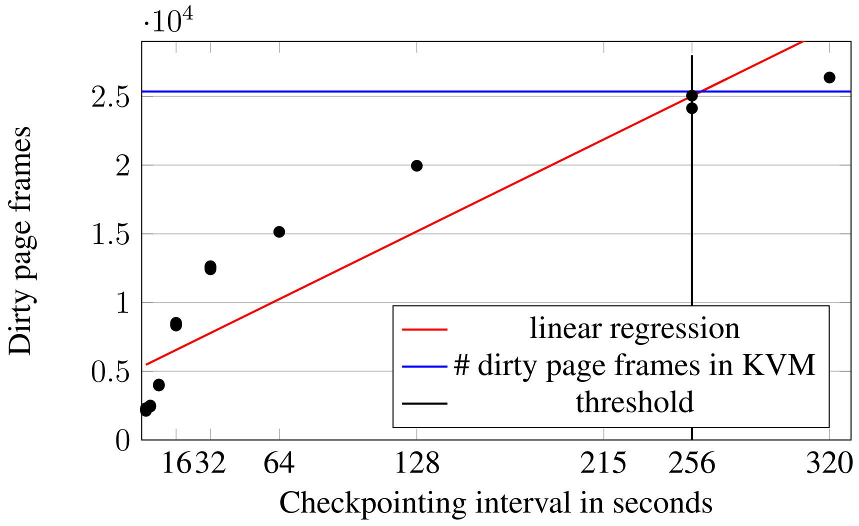

Figure 6.1: Average number of dirty page frames over 50 checkpoints during a kernel build with various checkpointing intervals and the threshold for proportionate data derived from checkpointing with virtualization and an interval of 2 seconds

Figure 6.1 shows the average number of modified page frames while running a kernel build and creating 50 checkpoints of an emulated system using various checkpointing intervals. A function approaching the dependency of checkpointing intervals and dirty page frames is found by applying a linear regression. Therefore an appropriate checkpointing interval for an emulated system can be found by taking the intersection of the average dirty page frames measured by Baudis in the virtualization and the regression curve that is generated using the tested checkpointing intervals.

Experiments showed that an interval of 256 seconds results in an average dirty page frames count between 24000 and 25000 per checkpoint that is matchable with the average dirty page frames count noticed by Baudis.

6.2 Evaluation Setup

The setup for data acquisition consists of a physical host system running a full system simulator and a tracing framework to store data. During data acquisition, running a simulation using QSimu and maintaining three open Simutrace connections plus running the Simutrace server itself, up to 6.2GB of RAM are consumend.

Component	Specification/Model
CPU	Intel(R) Core(TM) i7 CPU 920
Frequency	1.6Ghz – 2.67Ghz
Available Cores	4 plus hyperthreading
Main Memory	24GB
Storage	Intel SSDSA2M160, 160GB
Host OS	Ubuntu 15.04, 64 bit kernel
Simulator	QSimu 2.2.1 with hypercall interface
Storage facility (tracing)	Simutrace 3.1.1
Storage facility (analysis)	Simutrace 3.2.1
Guest OS	Linux Ubuntu 14.0.4.2
Guest kernel	version 4.1.0-rc2 with hypercall extension
Simulated CPU	single core x86_64
Simulation Main Memory	2GB

Table 6.2: Data acquisition and analysis configuration

Therefore, to serve the memory and computation requirements reliably, the host system contains an Intel i7 CPU with four physical cores a 2.67Ghz with hyperthreading and 24GB of RAM. One 160GB SSD arranges for fast persistent memory. It runs Ubuntu 15.04 with a generic 64 bit kernel. QSimu is used in version 2.2.1 and emulates a single core x86_64 system with 2GB of RAM running a Linux Ubuntu 14.04.2 with kernel version 4.1.0-rc2+ that is custumized by the introspection facilities mentioned in chapter 5. For recording of operating system and checkpointing events Simutrace version 3.1.1 is used, which includes the checkpointing mechanism of Eicher. However reading and merging Simutrace streams for analysis requires version 3.2.1 or higher as version 3.1.1 does not support stream multiplexing. Porting the checkpointing code to version 3.2.1 was resigned as created traces are compatible. Table 6.2 gives a review of the complete system configuration.

6.3 Main Memory Increment Semantics

The execution of a Linux kernel build, taking a checkpoint every 256 seconds results in an average dirty page frames count between 24000 and 25000 per checkpoint like expected because of the adaption of the checkpointing intervall, whereas the repeated execution of the 401.bzip2 benchmark only results in an average dirty page frames count between 13000 and 14000 (Figure 6.3).

Workload	**Avg. dirty page frames**	**Standard error**
Kernel build	24616	222
401.bzip2	13502	88

Table 6.3: Average dirty page frames per checkpoint using a checkpointing interval of 256 seconds

In Figure 6.2 the average allocation type distributions of both benchmarks are shown. The 401.bzip2 benchmark shows a lower percentage of dirty page frames assigned to page cache than the kernel build. This can be explained by the fact that once the processed files are available in the page cache they are not loaded again until the page cache is flushed after every 10th run of the execution.

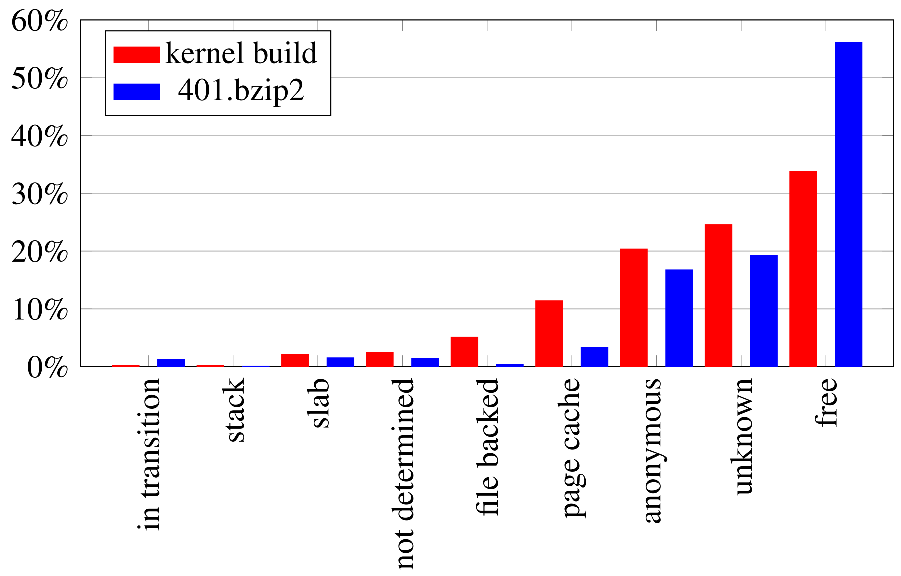

Figure 6.2: Average distribution of allocation types of dirty page frames per checkpoint during runs of three kernel builds and three 401.bzip2 benchmarks, taking 50 checkpoints

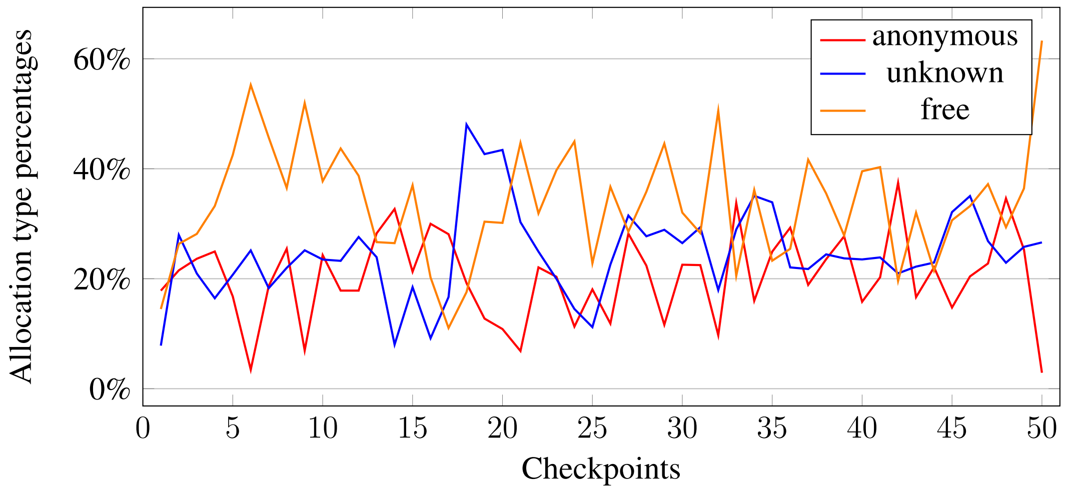

Figure 6.3: Average allocation type distribution for 50 checkpoints during multiple kernel builds. Only allocation types with average percentages greater than 15% are plotted.

Both workloads show a high percentage of page frames with unknown, hence page frames that are not at all traced by installed hypercalls. Therefore, the Linux kernel seems to often assign page frames without mapping them into a process' address space, as hypercalls are installed in all physical to virtual page mapping establishing kernel facilities.

Nevertheless, the most frequent page frames that are saved in a checkpoint, are page frames that were already freed by the buddy allocator thus are available for reallocation. This means the page frames were allocated and modified in between two checkpoints and freed again by the buddy allocator before the creation of the next checkpoint.

To further analyze allocation types, especially page frames marked as free, the allocation type percentages of dirty page frames per checkpoint during both workloads are plotted (Figure 6.3 and Figure 6.4). For simplicity allocation types whose percentages are on average less than 15% are omitted. When examining the plots of anonymous and free page frame percentages over time during a kernel build, there is a negative correlation recognizable. These assumption can be confirmed by calculating the Pearson correlation coefficent (PCC).

The PCC r_{xy} describes the linear dependency between two variables x and y. Its value ranges from -1 to 1. If $r_{xy} = -1$ the correlation between x and y is a negative linear correlation, this means all data points are lying on a line for which y increases as x decreases or the other way around. If $r_{xy} = 0$ there is no linear correlation and if $r_{xy} = 1$ it is a positive linear correlation with all data points lying on a line for which y increases as x increases or y decreases as x

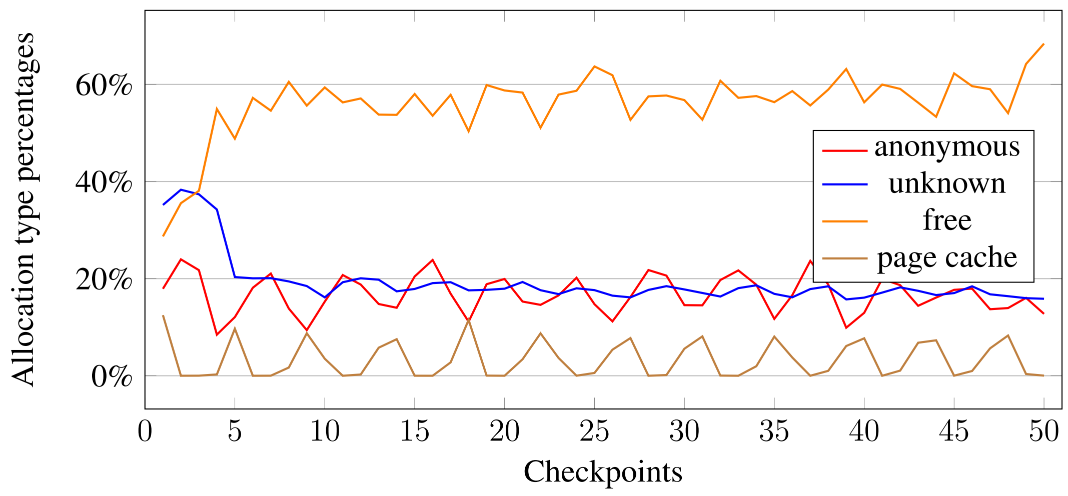

Figure 6.4: Average allocation type distribution for 50 checkpoints during multiple runs of the 401.bzip2 benchmark. Only allocation types with average percentages greater than 15% are plotted (except page cache).

decreases. Hence the closer the PCC of two variables is to -1 or 1 the higher is the correlation between both variables. The p-value, which is calculated with the PCC, indicates the statistical significance of a correlation. A p-value $p > 0.05$ indicates a statistical insignificant correlation.

The PCC of anonymous and free page frame percentages over time during a kernel build is $r_{xy} = -0.69$ with a p-value of $p = 0.235 \cdot 10^{-7}$, which indicates a moderate correlation with high statistical significance. Consequently it is valid to conclude that most page frames marked as free during a kernel build were allocated as anonymous pages frames before.

The plots of the 401.zip2 benchmark are less spread than observed in the plots of the kernel build. As the setup of the 401.bzip2 benchmark flushes the page cache periodically, the plot of the page cache allocation type reflects the periodic flushing by oszillating between 0% and about 10%. The peaks represent phases after flushing the page cache when it is filled again.

As the benchmark is working on the same files over and over again the distinct regions between two zero-points have to have a similar surface area, representing the size of the processed file, namely 8.07MB. The surface area is on average 2157.04 with a standard deviation of 129.07, which corresponds to an average filesize of 8.84MB with a standard deviation of 0.52MB. The average difference of 0.77MB between processed files filesize and the average number of megabytes allocated by the page cache may be related to bzip2 itself.

Additionaly the plots of anonymous and page cache page frames indicate a negativ correlation ($r_{xy} = -0.64, p = 0.549 \cdot 10^{-6}$). A minimum of the page cache allocation type indicates the availabilty of all files in memory. Whereas a peak in anonymous page frames denotes processing of files, because data processing requires private memory to work with.

Further analysis of unkown page frames, by looking into process affinity of allocation types, showed that 63% of all page frames marked with unknown were allocated when the GNU C Compiler context was active. If differences between the gradients of the unknown plot in the kernel build and the 401.bzip2 benchmark are taken into account, it can be assumed that the kernel build is triggering particular, main memory changing operating system mechanisms more often than the 401.bzip2 benchmark.

6.3.1 Semantics of Duplicate Page Frames

Section 6.3 presented a general analysis of semantics of dirty page frames during incremental checkpointing, providing an informative basis. Further analysis is going to examine duplicate page frames and their semantic characteristics.

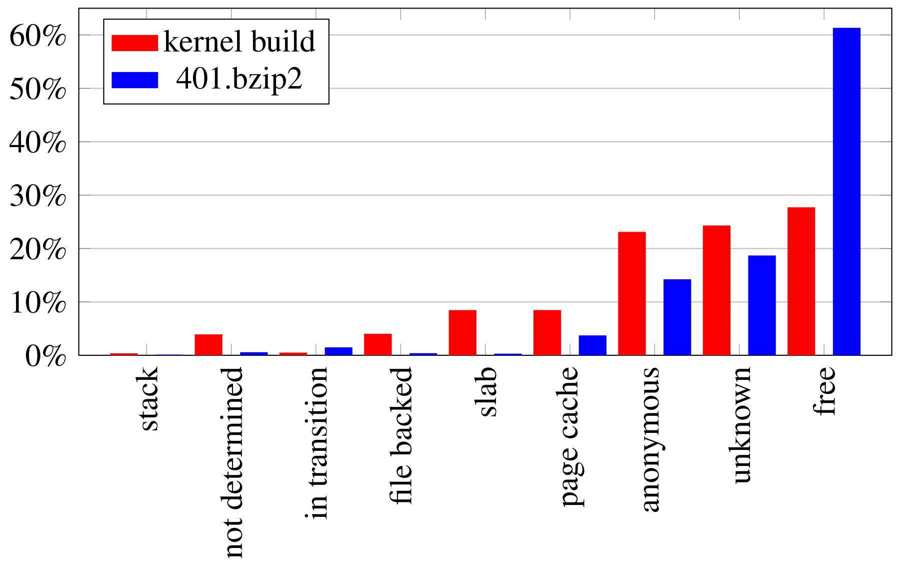

Figure 6.5: Average distribution of allocation types of deduplicated page frames per checkpoint during runs of three kernel builds and three 401.bzip2 benchmarks, taking 50 checkpoints

Baudis [5] measured on average 17% duplicates during multiple kernel builds and on average 42% during multiple runs of the 401.bzip2 benchmark. The setup of the 401.bzip2 benchmark in this thesis generates an even higher average deduplication rate of 85%. The result is not surprising as the benchmark runs repeat-

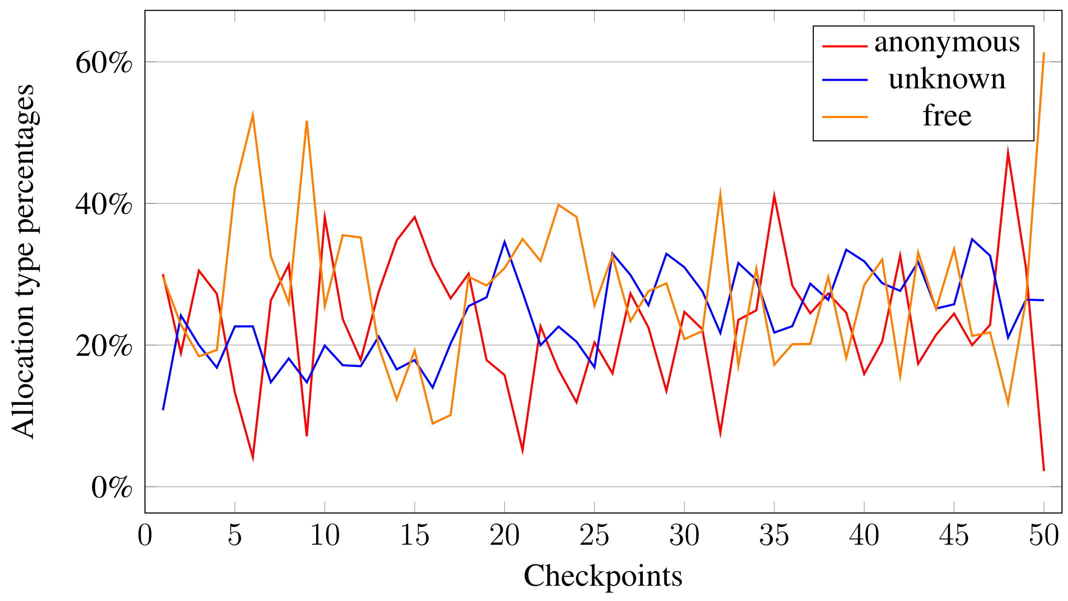

Figure 6.6: Average allocation type distribution of duplicates for 50 checkpoints during multiple kernel builds. Only averages greater than 15% are plotted.

edly over the course of 50 checkpoints. However, the average duplication rate during multiple kernel builds measured in this thesis is just 12%.

The evaluation of the average allocation type distribution (Figure 6.5) of page frame duplicates over 50 checkpoints of both workloads shows a similar pattern like the distribution of the dirty page frames stated in 6.3.

Main memory duplicates arising from a Linux kernel build mainly consist of page frames that are marked as free, are allocated as anonymous memory or have not been traced by installed hypercalls. Despite past work, which state the page cache as main source of page frame duplicates, only 8% of page frame duplicates in incremental checkpoints during a kernel build have their origin in the page cache. If it is taken into account that on average 11% of dirty page frames were allocated by the page cache, a possible explanation for less page cache duplicates is the relatively linear behavior of the page cache during a kernel build. Once files are loaded, they are unlikely to beeing reloaded, hence produce no duplicates in incremental checkpoints.

The 401.bzip2 benchmark indicates even less page frame duplicates located in the page cache, as the 401.bzip2 benchmark setup can be divided in three reappearing, overlapping stages with low duplication potential. First of all the page cache is filled with predetermined data, which is unlikely to produce duplicate page frames as loaded files usualy differ from each other. Afterwards, loaded files are processed and page cache modifications recede. Finally, the page cache

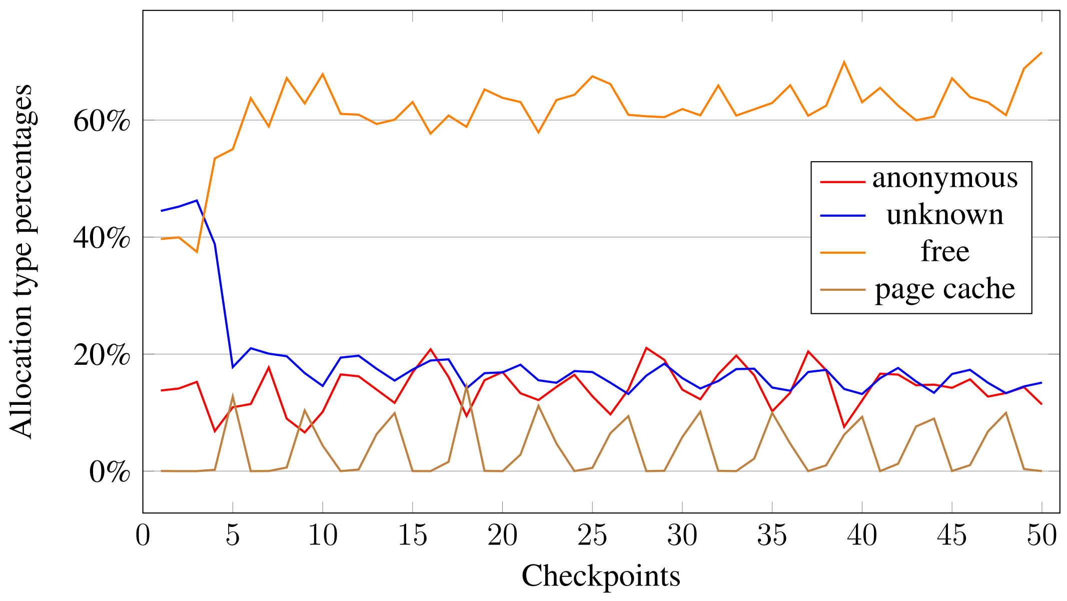

Figure 6.7: Average allocation type distribution of duplicates for 50 checkpoints during multiple runs of the 401.bzip2 benchmark. Only averages greater than 15% are plotted.

is flushed and all files, hence deduplication potential, are removed and the page cache is refilled again. This process is represented, by the oscillating curve of the plot of the page cache in the graphs of allocation type distributions of dirty and deduplicated page frames over the course of 50 checkpoints (see Figures 6.4 and 6.7).

In general the plots of allocation type distributions of duplicates over time (Figures 6.6 and 6.6) and the plots of allocation type distributions of dirty page frames over time are much alike (Figures 6.3 and 6.4).

Therefore the obvious negative correlation of duplicate page frames categorized as free and such categorized as anonymous memory is once again described by the Pearson correlation. It results in $r_{xy} = -0.77$ with a p-value of $p = 0.59 \cdot 10^{-10}$. The correlation of duplicate free and anonymous page frames is even more significant than the correlation of dirty page frames marked as free and anonymous respectively ($r_{xy} = -0.69, p = 0.235 \cdot 10^{-7}$).

For a more detailed evaluation of duplicate page frame semantics, the analysis tool procsses the data hashes of the deduplication process of the checkpointing mechanism. By counting the number of their reoccurrences it is possible to obtain an idea of memory contents that probably result in a big amount of duplicates.

Figure 6.8 shows the average data hash count of three kernel builds per allocation type. It is noticeable that each allocation type has a peak in counts at the

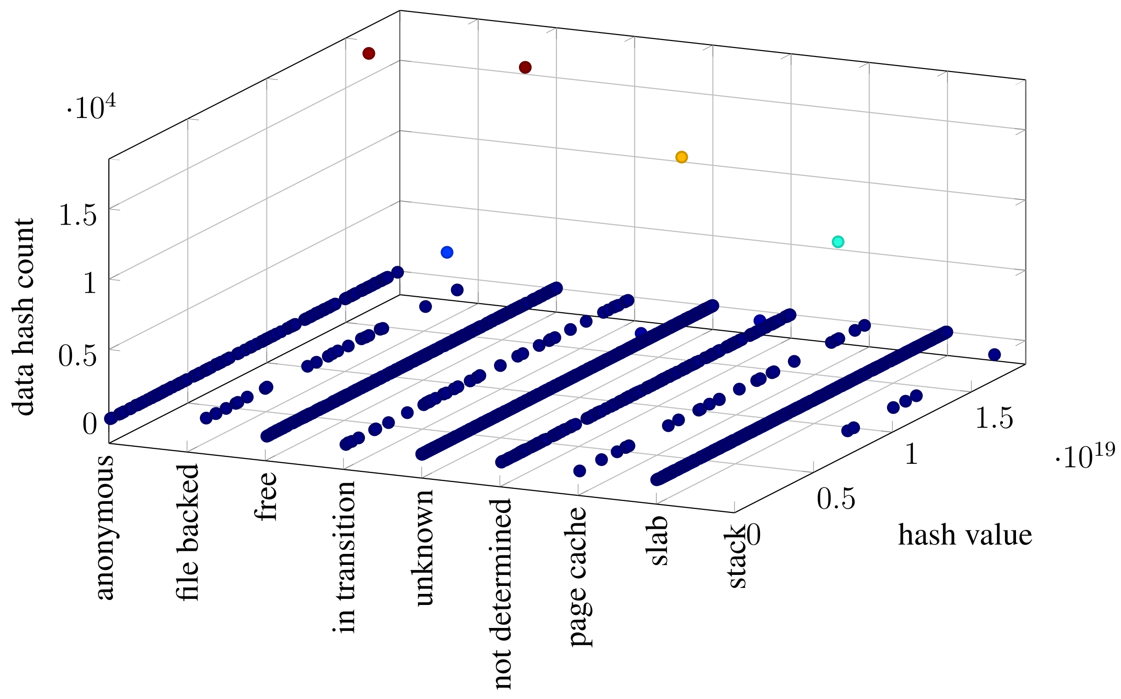

Figure 6.8: Count of data hash reoccurence per allocation type. Only data hashes with a reoccurence count greater than 10 are showed

data hash 0xe488d4952d64521f. As there are not as many possible explanations for this many reappearing identical page frames, the first assumption is that the data hash represents the page frame entirely filled with zeroes. This page frame is henceforward referred to as *zero page frame* or simply zero page. This assumption is confirmed as calculating the data hash over a zero page results in the exact same value.

6.3.2 Zero Pages

The outcome of the allocation type and data hash analysis regarding the zero page frames raises the question for their share in page frame duplicates of incremental checkpoints. Therefore their contribution to duplicate page frames per allocation type is calculated for both workloads. Table 6.4 and Table 6.5 show the number of reappearing data hashes, which were counted seperatly for each allocation type. In this case reappearing means that only data hashes that occured more than once were counted. The data hashes were counted regardless of the checkpoints they occured at. The total number of reappearings is the sum of reappearings of data hashes of all allocation types.

Allocation type	Number of reappearing data hashes	Percentage zero page hash
anonymous	23347	70,91%
file backed	3831	80,59%
free	38387	43,91%
page cache	10343	66,99%
slab	11874	0,32%
stack	242	55,84%
in transition	523	7,47%
not determined	5635	12,31%
unknown	37046	31,55%
total	131226	42,76%

Table 6.4: Number of reappearing data hashes, hence page frame duplicates, of kernel builds and which portion of it belongs to zero page frames

The total number of reappearing data hashes, hence page frame duplicates, of the 401.bzip2 benchmark is higher than the number measured during kernel builds, because the 401.bzip2 benchmark has a higher duplication rate in general. Obviously both workloads show a significantly high percentage of zero page frames among duplicates.

An explanation for at least a fragment of the zero pages that are anonymous or free, is given by programms that actively set allocated memory to zero. This is often the case when buffers are initialized. However this does not explain the high percentage of zero pages in file backed page frames or page frames that are assigned to the page cache or slab.

A detailed evaluation of zero pages and their origin and lifetime in memory is not within the scope of this thesis.

6.4 Conclusion

The evaluation showed that between 30% and 60% of dirty and duplicate page frames in incremental checkpoints are marked free. Furthermore an correlation between free page frames and anonymous page frames during a kernel build was detected. Therefore most page frames that are marked as free during a kernel build were allocated as anonymous page frames before.

Allocation type	**Number of reappearing data hashes**	**Percentage zero page hash**
anonymous	81783	31,12%
file backed	1748	10,8%
free	352285	61,86%
page cache	25822	5,01%
slab	1266	0,06%
stack	130	42,67%
in transition	6646	80,22%
not determined	2969	9,92%
unknown	102410	44,05%
total	575059	51,38%

Table 6.5: Number of reappearing data hashes, hence page frame duplicates, of 401.bzip2 benchmark and which portion of it belongs to zero page frames

A more detailed evaluation of duplicate page frame semantics, by counting data hashes per allocation type, presented zero page frames as main origin of page frame duplication in incremental checkpoints. Namely 43% of duplicates during a Linux kernel build and 51% during the 401.bzip2 benchmark were zero pages.

It has to be determined if incremental checkpointing mechansims like implemented by Baudis [5] and Eicher [8] could pass on deduplication and only exclude zero pages as they represent between 40% and 50% of duplicate page frames depending on the workload. An approach without deduplication but excluded zero pages could save time that is used to calculate data hashes and deduplicate page frames. Alternatively a combination of both approaches is imaginable as well. By modifying the checkpointing mechanism to skip zero pages when sending dirty page frames to Simutrace, the data amount and processing time could be reduced noticeably.

The checkpointing mechanism of Aderholdt et al. [2] uses introspection to locate unused (free) page frames, that are excluded from checkpoints. As between 44% and 64% of unused page frames are zero pages, it raises the question if the introspection approach could be improved by ignoring zero pages or if it is sufficient to scan the VM for zero pages to exclude and disregard the introspection.

Chapter 7

Conclusion

The checkpoints that were analyzed in this thesis originate from a framework that targets a performance increase of full system simulation by the use of parallelization. The checkpoints, which are created in a virtualized system, serve as starting points for nodes, that simulate simultaneously. The checkpoint creation has to meet specific requirements concerning duration of checkpoint creation and data amount in order to provide a satisfying simulation speedup. Therefore, an incremental checkpointing mechanism with deduplication of page frames and disk sectors was developed by Baudis [5] and Eicher [8].

By acquiring semantical information of the virtual machine, it was possible to perform a detailed analysis of page frames deduplicated in incremental checkpoints. The virtual machine was inspected using operating system introspection. This technique provides a good trade-off between usability and effect on the system running in the virtual machine.The implemented introspection infrastructure integrates into an existing tracing framework and interfaces with the QEMU full system simulator.

The evaluation of collected semantical information showed characteristics of dirty page frames during incremental checkpointing and delivered insights into the source of page frame duplicates in incremental checkpoints. It showed that on average between 30% and 60% of dirty and duplicate page frames in incremental checkpoints are marked free, which is the majority of page frames. Furthermore there is a correlation between anonymous and free page frames during a Linux kernel build. Therefore most page frames that are marked as free during a kernel build were allocated as anonymous page frames before.

Analysis of reoccurences of data hashes revealed zero page frames to be the main origin of page frame duplication. Between 40% and 50% of duplicate page frames are zero pages. This information can be used to improve existing checkpointing mechanisms. The checkpointing mechanism implemented by Baudis [5] and Eicher [8] can be modified to skip zero pages when sending page frames to

Simutrace for deduplication and storage. This would reduce the data amount noticeably.

The checkpointing mechanism presented by Aderholdt et al. [2] uses introspection to locate free page frames and exclude them from checkpoints. As between 44% and 64% of free page frames are zero pages there is also potential for improvement.

7.1 Future Work

About 20% of dirty page frames were not traced by the hypercalls that were installed in the guest systems operating system kernel. Therefore it is required to complete the operating system introspection used in this thesis, so the conclusions of this work can be stated more precisely.

The knowledge of zero page frames beeing the main source of duplicate page frames, can be used to implement an improved introspection based approach to further reduce the data amount of incremental checkpoints and therefore increase their use for parallel full system simulation. An detailed analysis of the occurence and lifetime of zero page frames can be helpful in this context as well.

The existing incremental checkpointing mechanism of Baudis [5] and Eicher [8] can be improved by considering exclusion zero pages as well.

Bibliography

[1] Intel 64 and ia-32 architectures software developer's manual.

[2] F. Aderholdt, Fang Han, S.L. Scott, and T. Naughton. Efficient checkpointing of virtual machines using virtual machine introspection. In *Cluster, Cloud and Grid Computing (CCGrid), 2014 14th IEEE/ACM International Symposium on*, pages 414–423, May 2014.

[3] Eduardo Argollo, Ayose Falcón, Paolo Faraboschi, Matteo Monchiero, and Daniel Ortega. Cotson: Infrastructure for full system simulation. HP Laboratories, 2009.

[4] Sean Barker, Timothy Wood*, Prashant Shenoy, and Ramesh Sitaraman. An empirical study of memory sharing in virtual machines. University of Massachusetts Amherst, *The George Washington University.

[5] Nikolai Baudis. Deduplicating virtual machine checkpoints for distributed system simulation. Bachelor thesis, System Architecture Group, Karlsruhe Institute of Technology (KIT), Germany, November2 2013. `http://os.itec.kit.edu/`.

[6] Bellard and Fabrice. Qemu, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track*, pages 41–46. USENIX, 2005. `http://dblp.uni-trier.de/db/conf/usenix/usenix2005f.html#Bellard05`.

[7] Daniel P. Bovet and Marco Cesati. *Understanding the Linux Kernel*. O'Reilly, 2005.

[8] Bastian Eicher. Virtual machine checkpoint storage and distribution for simuboost. Master thesis, System Architecture Group, Karlsruhe Institute of Technology (KIT), Germany, September30 2015. Will be published at `http://os.ibds.kit.edu/`.

[9] Jakob Engblom. Full-system simulation. European Summer School on Embedded Systems, 2003.

[10] Thorsten Gröninger. On statistical properties of duplicate memory pages. Diploma thesis, System Architecture Group, Karlsruhe Institute of Technology (KIT), Germany, October31 2013. `http://os.ibds.kit.edu/`.

[11] Zhongshu Gu, Zhui Deng, Dongyan Xu, and Xuxian Jiang. Process implanting: A new active introspection framework for virtualization. In *SRDS*, pages 147–156. IEEE Computer Society, 2011. `http://dblp.uni-trier.de/db/conf/srds/srds2011.html#GuDXJ11`.

[12] Jonas Julino. Lightweight introspection for full system simulations. Diploma thesis, System Architecture Group, Karlsruhe Institute of Technology (KIT), Germany, March1 2014. `http://os.itec.kit.edu/`.

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

[14] Peter S. Magnuson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hallberg, Johan Hogberg, Fredrik Larsson, Andreas Morstedt, and Bengt Werner. Simics: A full system simulation platform. 2002.

[15] Marc Rittinghaus. Runtime benefits of memory deduplication, July5 2012. `http://os.ibds.kit.edu/`.

[16] Marc Rittinghaus, Thorsten Groeninger, and Frank Bellosa. Simutrace: A toolkit for full system memory tracing. Operating Systems Group Karlsruhe Institute of Technology (KIT), 2015.

[17] Marc Rittinghaus, Konrad Miller, Marius Hillenbrand, and Frank Bellosa. Simuboost: Scalable parallelization of functional system simulation. System Architecture Group Karlsruhe Institute of Technology (KIT), 2013.

[18] Qemu user documentation. http://qemu.weilnetz.de/qemu-doc.html.

[19] Qemu/debugging with qemu. https://en.wikibooks.org/wiki/QEMU/Debugging_with_QEMU

[20] Qemu internals. http://qemu.weilnetz.de/qemu-tech.html.

[21] Simutrace. http://simutrace.org.