KU LEUVEN

ARENBERG DOCTORAL SCHOOL
Faculty of Engineering Technology

Elevating Multithreading Further into the Cloud

Evaluation and Amelioration of Hardware-Assisted Virtualization for Multithreaded Applications in X86

Stijn Schildermans

Supervisors:
Prof. dr. K. Aerts
Prof. dr. ir. T. Schrijvers
Prof. dr. X. Ding
(New Jersey Institute of Technology)

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor of Engineering Technology (PhD): Electronics-ICT

June 2022

Elevating Multithreading Further into the Cloud

Evaluation and Amelioration of Hardware-Assisted Virtualization for Multithreaded Applications in X86

Stijn SCHILDERMANS

Examination committee:
Prof. dr. ir. M. Vergauwen, chair
Prof. dr. K. Aerts, supervisor
Prof. dr. ir. T. Schrijvers, supervisor
Prof. dr. X. Ding, supervisor
(New Jersey Institute of Technology)
Prof. dr. ir. D. Weyns
Dr. ing. L. Vandeurzen
Prof. dr. ir. M. Verhelst
Dr. L. Cuypers
(Commeto)
Prof. dr. J. Shan
(Hofstra University)

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor of Engineering Technology (PhD): Electronics-ICT

June 2022

Preface

The story of this Ph. D. project starts with that of my Master's thesis, in which I explored the potential of functional programming in various cloud environments. The main conclusion of this work was that the functional programming style lends itself very well to the cloud, but at the time the severely limited support cloud platforms offered for functional programming languages inhibited developers from fully exploiting this natural synergy. This finding originally led to my Ph. D. project, which aimed to develop a platform and accompanying software framework that allowed practitioners to fully harness the potential of functional programming in the cloud.

Naturally, the findings of my Master's thesis would serve as the outset for developing the platform and framework described above. As such, the first goal of my Ph. D. was to understand the underlying mechanisms that led to these findings. Early on in this process, my attention was drawn to a particularly interesting observation my Master's thesis made, being that multithreaded functional programs outperformed their traditional object-oriented counterparts in the cloud. Soon I found myself combing through system software and hardware manuals in search for an explanation. I discovered that virtualizing multithreaded applications—functional or not—is in fact far from trivial and doing so may to this day induce severe performance issues. This led me to realize that the observation I made during my Master's thesis is in fact but a symptom of a much broader problem, namely efficiently virtualizing multithreaded applications. This realization compelled me to shift the focus of my Ph. D. from functional programming to multithreading in a cloud context, eventually leading to the dissertation before you right now.

For me personally, the Ph. D. project documented in this dissertation means much more than simply a set of scientific contributions. For the past five years, it was my principal goal in life and permeated a large part thereof. It forced me to grow from a shy student used to doing what he is told to an assertive researcher proactively looking for and tackling new challenges. Needless to say, this journey

was not without moments of doubt, desillusion and frustration. However, looking back on it all, I am immensely proud that I persevered, knowing that aside from scientific contributions, this project has also brought me enormous fulfillment and personal enrichment in the form of unique experiences, knowledge and character development.

Having admitted that I experienced this work as challenging at times, it is no more than fitting to thank the persons enabling me to bring it to a successful conclusion regardless. Foremost in this list is prof. dr. Kris Aerts, who granted me the opportunity and funding to perform this Ph. D. and acted as my main supervisor throughout. Additionally, I would like to thank prof. dr. Xiaoning Ding and prof. dr. Jianchen Shan for providing me with invaluable technical advise when working on several of the publications upon which this dissertation is based and Hofstra University and KU Leuven for providing the infrastructure necessary to perform the experiments supporting many of the findings presented in this work. Furthermore, I would like to thank prof. Ding once again in conjunction with the New Jersey Institute of Technology, the Flemish Research Foundation and KU Leuven for making it possible for me to travel to the United States of America for 4.5 months within the context of this Ph. D. project; to this day the most enriching experience of my life. Finally, I would like to thank my parents—Georges Schildermans and Godelieve Billiau—and my girlfriend—Merel Vaes—for their unquestioning emotional support throughout all these years.

Abstract

Due to the surging popularity of cloud computing on one hand and the emergence of numerous novel, innately parallel workloads on the other, executing multithreaded applications in a virtualized setting has become common practice in industry. However, multithreading is known to be highly susceptible to severe performance degradation in virtualized environments. In response, virtualization technologies have evolved rapidly over the years; to the point of virtual machines being considered comparable to their physical counterparts in terms of performance. Precisely because of the rapidity of this evolution however, empirical evidence supporting this consensus is limited at best. Moreover, the crippling levels of performance degradation described in literature less than a decade old suggest that—rapid improvements notwithstanding—it is more than likely that several challenges still remain in this regard. Both identifying and addressing these challenges are the main goals of this work.

Because virtualization is a very broad term, this manuscript commences by describing the virtualization process in general and situating the scope of this Ph. D. project within this broad landscape. Next, it performs a much needed assessment of the state of the art by measuring virtualization overhead for a variety of multithreaded applications through controlled experiments, after first formally defining what exactly virtualization overhead entails within this context. A reflection on potential mitigation techniques for the remaining challenges these experiments lay bare follows. Finally, it refines, implements and evaluates three of the most promising of these techniques, carefully selected to each target a distinct level of the system stack so that they are complementary to one another.

This dissertation makes clear that virtualization overhead is a multifaceted phenomenon, in essence exclusively internal to the system in the form of reduced resource efficiency. Nevertheless, this reduction in resource efficiency may be observable externally in the form of a reduction in temporal efficiency. In particular for multithreaded applications, these system and application effects

may differ signifficantly in magnitude. Specifically, this work shows that these effects may still amount to respectively 170% and 80% for multithreaded applications in a state-of-the-art virtualized environment. Although these numbers suggest that much work remains to be done, the complementary mitigation techniques this work elaborates on represent a solid step in the right direction. In particular, chapter 6 presents 'virtual scheduler ticks' as a means to address excessive virtualization overhead caused by rapid switches between idle and active vCPU states in tickless systems by paravirtualizing the scheduler tick, improving performance by up to 15%. Furthermore, chapter 7 addresses TLB shootdown overhead induced by rapidly resizing application memory space, resulting in the concept of 'global hysteresis', which yields performance gains of up to 45%. Finally, chapter 8 outlines a series of guidelines application developers may follow to minimize the likelihood of their code suffering significant virtualization overhead. Although the effect of applying these guidelines depends greatly on the nature of the application, the proof of concept included in this manuscript achieves performance improvements of up to 40%.

Beknopte samenvatting

Omwille van de toenemende populariteit van cloud computing alsook de opkomst van verschillende nieuwe, van nature parallelle toepassingen is het uitvoeren van applicaties die gebruik maken van multithreading in een gevirtualiseerde context een standaardpraktijk geworden in de industrie. Desalniettemin staat multithreading erom bekend zeer gevoelig te zijn voor performantieproblemen in een gevirtualiseerde omgeving. Omwille hiervan zijn virtualisatietechnologieën doorheen de jaren aan een hoog tempo geëvolueerd; zelfs zodanig dat virtuele machines de dag van vandaag gelijkwaardig worden geacht aan hun fysieke tegenhangers wat betreft performantie. Precies door het hoge tempo van deze evolutie is empirisch bewijs ter ondersteuning van deze consensus echter op zijn zachtst gezegd beperkt. Daarenboven suggereren de enorme performantiedegradaties beschreven in literatuur die nog maar enkele jaren oud is dat er op dit gebied meer dan waarschijnlijk nog tal van uitdagingen overblijven. De voornaamste doelstellingen van dit werk zijn dan ook het identificeren en het aanpakken van deze uitdagingen.

Omdat virtualisatie een zeer breed begrip is vangt deze thesis aan met een beschrijving van het virtualisatieproces in het algemeen en een afbakening van het gebied dat dit doctoraatsproject bestrijkt binnen dit brede landschap. Vervolgens gaat dit werk de stand van zaken binnen dit gebied na door aan de hand van experimenten virtualisatie-overhead op te meten voor een brede waaier aan applicaties die gebruik maken van multithreading, na eerst formeel te definiëren wat virtualisatie-overhead eigenlijk inhoudt binnen deze context. Hierop volgt een reflectie over mogelijke oplossingen voor de resterende problemen die deze experimenten onthullen. Ten slotte wijdt dit werk uit over drie van de meest veelbelovende dezer mogelijke oplossingen, die aandachtig geselecteerd zijn zodat ze elk betrekking hebben op een verschillende laag in de systeem stack en elkaar dus automatisch aanvullen.

Dit proefschrift maakt duidelijk dat virtualisatie-overhead uit vele facetten bestaat en in eerste instantie een louter intern systeemfenomeen is dat

zich manifesteert in de vorm van verminderde systeembronefficiëntie. Deze verminderde systeembronefficiëntie kan op zijn beurt echter extern worden waargenomen in de vorm van verminderde tijdsefficiëntie. Specifiek voor applicaties die gebruik maken van multithreading kunnen de groottes van deze systeem- en applicatie-effecten sterk van elkaar verschillen. Concreet toont dit werk aan dat deze effecten nog steeds respectievelijk 170% en 80% kunnen bedragen voor applicaties die gebruik maken van multithreading in zelfs de modernste gevirtualiseerde omgevingen. Hoewel deze resultaten suggereren dat er nog veel werk voor de boeg ligt vormen de technieken die dit proefschrift naar voor draagt een aanzienlijke stap in de goede richting. Specifiek stelt hoofdstuk 6 het concept van 'virtual scheduler ticks' voor als een manier om buitensporige virtualisatie-overhead veroorzaakt door snelle overgangen tussen actieve en inactieve vCPU toestanden in tickless systemen tegen te gaan door paravirtualisatie toe te passen op de scheduler tick. Verder pakt hoofdstuk 7 TLB shootdown overhead veroorzaakt door aan een hoog tempo de geheugenruimte van applicaties in grootte aan te passen aan, wat leidt tot het concept van 'global hysteresis' wat op zijn beurt performantiewinsten tot 45% bewerkstelligt. Ten slotte beschrijft hoofdstuk 8 een reeks richtlijnen voor programmeurs met als doel de kans dat zij code schrijven die significante virtualisatie-overhead veroorzaakt te minimaliseren. Hoewel het effect van deze richtlijnen sterk afhankelijk is van de specifieke applicatie waarop ze worden toegepast, bereikt de bijgevoegde demonstratieve applicatie een performantiewinst van 40% na toepassing van deze richtlijnen.

List of Abbreviations

ABI Application Binary Interface. 16, 17, 30

AI Artificial Intelligence. 2, 116

API Application Programming Interface. 23, 109, 110, 121, 153, 168, 173

APIC Advanced Programmable Interrupt Controller. 25, 75

BWW Blocked Waiter Wakeup. 44, 73, 130

CPI Cycles Per Instruction. 69, 70

CPU Central Processing Unit. 4, 8, 10, 14, 17–20, 23–25, 38–41, 43–48, 53, 55, 57, 61, 62, 64, 67, 68, 70–73, 75, 77, 78, 80, 81, 88–91, 94–96, 98, 109–112, 114, 123, 128, 130, 136–140, 148, 149, 154, 156, 157, 160, 162, 163, 167, 181, 184, 186, 187

DAS Directly Attached Storage. 33

DID Direct Interrupt Delivery. 130

DMA Direct Memory Addressing. 21, 23, 24, 41

EIE External Interrupt Exiting. 130

EPT Extended Page Table. 20

FIFO First-In-First-Out. 42

GB GigaBytes. 186

GPA Guest-Physical Address. 19, 20

GPGPU General-Purpose Graphics Processing Unit. 116

GPU Graphics Processing Unit. 25

GVA Guest-Virtual Address. 19, 20

HPA Host-Physical Address. 19, 20

HPC High-Performance Computing. 3, 4, 59

Hz Hertz. 110, 116, 154

I/O Input/Output. ix, 2, 10, 21–25, 31, 38, 39, 41, 46, 48, 60–62, 67, 92, 114, 116, 118, 122, 128–131

ICR Interrupt Command Register. 44, 45, 149

ID Identifier. 89, 99

IoT Internet of Things. 2

IP Internet Protocol. 34

IPI Inter-Processor Interrupt. 43–45, 73, 75, 76, 79, 81, 86, 89, 92, 98, 99, 137–140, 143, 149, 187, 188

ISA Instruction Set Architecture. 8, 14, 15

IT Information Technology. 1, 36

JIT Just-In-Time. 30

JRE Java Runtime Environment. 30

JVM Java Virtual Machine. 30, 96

kB Kilobytes. 128, 135, 141, 143, 178

LAN Local Area Network. 35

LAPIC Local Advanced Programmable Interrupt Controller. 110, 112

LBA Logical Block Addressing. 32

LHP Lock Holder Preemption. 42, 44, 45, 76, 79, 94, 95, 97, 130

LWP Lock Waiter Preemption. 42, 43, 94, 95, 97, 130

MB MegaByte(s). 31, 135, 141, 142, 144, 159, 184, 186

MMIO Memory-Mapped Input/Output. 21, 23

MMU Memory Management Unit. 18–20

ms millisecond(s). 38, 116

MSR Model-Specific Register. 44, 45, 73, 75, 76, 94, 112, 113, 123, 130, 149

NAS Network-Attached Storage. 33, 34

NIC Network Interface Card. 25

NUMA Non-Uniform Memory Access. 45–47, 54, 57, 60–62, 64, 67, 69–71, 78, 81, 83, 85–87, 100–105, 126, 134, 135, 137, 139, 140, 148, 149, 156, 157, 160, 178, 179, 182, 186, 188, 191

NVMe Non-Volatile Memory express. 116

OC OverCommitted. 47, 49, 52, 61, 62, 64, 67, 69, 71, 72, 75–82, 87, 94, 95, 113, 122, 123

OC_2 OverCommitted base two. 49, 52, 62, 64, 69, 77–80, 94

OPS Operations per Second. 55, 56

OS Operating System. 9–13, 16, 18–20, 26, 27, 29–33, 42–44, 47, 48, 61, 64, 75, 76, 80, 81, 88, 91, 97, 109–111, 122, 130, 138, 141, 144–146, 155, 163, 168, 178, 186

PCIe Peripheral Component Interconnect Express. 25

pCPU Physical Central Processing Unit. 14, 24, 44, 46, 47, 62, 89, 91, 101, 104, 116, 117, 130

PF Pause Filter. 43, 77

Ph. D. Doctor of Philosophy. 3, 5, 49, 81, 106, 166, 179, 189, 191, 193

PID Process Identifier. 26

PLE Pause Loop Exiting. 43, 76, 77, 79, 86, 94–97, 107, 123

PTE Page Table Entry. 18, 20, 44, 135, 136

RAID Reduntant Array of Independent Disks. 33

RAM Random Access Memory. 10, 22, 27

RCU Read-Copy-Update. 73, 112, 117, 122

RDT Resource Director Technology. 41

SAN Storage Area Network. 33, 34

SDK Software Development Kit. 174

SDN Software-Defined Networking. 34

SDS Software-Defined Storage. 34

SMP Symmetric MultiProcessing. 2, 109

SMT Symmetric Multithreading. 86, 91, 104

SR-IOV Single Root Input/Output Virtualization. 25, 67, 128

SSD Solid State Drive. 128

TLB Translation Lookaside Buffer. 18–20, 44, 45, 76, 79, 81, 87, 98–100, 106, 130, 133–141, 143, 144, 146–149, 153, 154, 156, 157, 160, 162–164, 176, 177, 181–183, 185, 187

TPU Tensor Processing Unit. 116

TSC Time Stamp Counter. 73, 112

UC UnderCommitted. 47, 49, 62, 64, 67, 69–72, 75, 76, 78, 79, 87, 88, 92, 93, 101, 103, 122, 123

vCPU Virtual Central Processing Unit. 14, 24, 25, 39, 42–46, 60–62, 64–67, 70, 72, 73, 75, 78–81, 86–93, 95, 97–99, 101–105, 112–119, 121, 122, 124, 126, 128–130, 186, 187

VIP Virtual Internet Protocol. 34

VIPT Virtually Indexed, Physically Tagged. 98

VLAN Virtual Local Area Network. 35

VM Virtual Machine. 8–12, 17, 19–27, 29–31, 37–43, 46–49, 60–62, 69, 72, 73, 80, 88, 89, 91, 97, 99–102, 104, 105, 112–119, 124, 126, 128, 130, 167, 182, 186, 192

VMCS Virtual Machine Control Structure. 17, 24, 130

VMM Virtual Machine Monitor. 10–25, 31, 39, 41–47, 72, 76, 79, 80, 87, 90, 91, 95, 99, 101, 102, 104, 105, 112, 113, 116, 117, 121, 130, 135, 186

VPN Virtual Private Network. 35

WAN Wide Area Network. 35

List of Symbols

$\delta\eta_r$	Reduction in Resource Efficiency
$\delta\eta_t$	Reduction in Temporal Efficiency
γ	Central Processing Unit Count
ω	Overhead Impact Factor
σ	Variance
C	Cycles
P	Physical System
S	System Settings
t	Wall Clock Application Execution Time
V	Virtual Machine
W	Workload

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

This brief introductory chapter outlines the context in which the research presented in this dissertation has been performed, derives the research problems addressed in this work from said context and establishes concrete objectives based on these problems. Finally, it provides a synopsis including a summary of the research papers on which this dissertation is based.

1.1 Context

Cloud computing is among the most impactful computing paradigms to emerge in decades. Since its initial formalization in 1997, it has grown to a leading software deployment model [1]. According to Eurostat, 36% of European businesses employed some form of cloud computing in 2020, up from 24% in 2018 [2]. This significant and growing corporate interest in and dependence on cloud computing is projected to continue to increase for years to come [3].

Although cloud services vary greatly in design and implementation, the common denominator among all of them is heavy use of virtualization [4]. This technology encompasses emulating information technology (IT) resources safely and efficiently, de facto instantiating (virtual) computing resources largely independently of the underlying physical infrastructure [5]. This in itself is not at all a new concept [6]. Over the five decades since its formal introduction, virtualizaton technology has become highly mature thanks to extensive efforts from academia and industry. Consequently, virtualized resources are these days expected to perform practically as well as physical ones [7].

The maturity of virtualization is undoubtedly a major driver of the accelerating adoption of cloud computing, since historically performance constraints originating at the virtualization infrastructure were among the main limitations of this novel paradigm [8]. On the other hand, as cloud computing is adopted for more and more diverse and demanding use cases such as artificial intelligence (AI), the internet of things (IoT) and big data, the limits of virtualization technology are continually being pushed. Thus, as much as improving virtualization performance is driving the adoption of cloud computing, the adoption of cloud computing is driving the need for ever more efficient and flexible virtualization technologies. Key to the continued success of cloud computing is the development of virtualization technology staying ahead of the growing demands of its adopters in this bilateral evolution.

1.2 Problem Statement

In spite of the increasing expectation of the contrary outlined above, even state-of-the-art virtualization techniques still struggle to efficiently virtualize certain system components and workloads. For example, input/output (I/O) devices are notoriously difficult to virtualize, implying that applications performing large amounts of I/O operations may still incur a significant performance penalty in a virtualized environment [9]. Even more problematic are multithreaded applications. Typical thread synchronization and data sharing constructs often require special handling in a virtualized environment, again inducing a severe performance penalty [10]. Moreover, entirely cost-free virtualization is nigh impossible, since as outlined in §1.1, virtualization entails emulating resources which are not (necessarily) physically present, which is almost invariably less efficient than directly employing said resources in physical form. Thus, optimizing virtualization technology may well prove to be an unending endeavor. It is therefore clear that even after half a century of progress there is still a strong need to further reduce the cost of the virtualization process.

The innate performance drawbacks of virtualization have been known since its inception [6]. Nevertheless, research efforts to ameliorate virtualization performance were limited during the first decades of its existence, likely because correctness and robustness were of greater concern. Additionally—or perhaps consequently—industrial applications of the technology were rare. This status quo changed radically in the beginning of the 21st century however, as powerful symmetric multiprocessing (SMP) servers and robust virtualization technology allowed for multiple virtualized systems to be hosted on a single physical platform, yielding significant cost savings [11]. Many solutions to long-standing challenges in the field have been proposed since, some of which have been

widely adopted [12, 13, 14, 15, 16]. However, largely due to the speed at which virtualization technology has evolved in recent years, the current state of the art regarding virtualization performance in an industrial context is unclear. Reliable empirical evidence for the efficacy of the many novel features sported by modern virtualization technology is lacking. Moreover, it is currently unclear which challenges remain to achieve truly efficient virtualization for all workloads, under all conditions.

From the above, the two principal problems this dissertation aims to address emerge:

- Despite great advancements in recent years, virtualization may still introduce a significant performance penalty for certain workloads;
- Both the nature and severity of the remaining challenges regarding virtualization performance are currently unclear.

1.3 Objectives

Virtualization is an immensely broad field, covering all kinds of system components and workloads [17]. Therefore, addressing the problems described in §1.2 in a general sense is infeasible within the context of a single Ph. D. dissertation. As such, the scope of this work must by practical necessity be limited to select virtualization technologies and workloads. Since the Ph. D. project documented in this dissertation has taken place within the faculty of engineering technology, industrial relevance was the primary concern in this selection process. Below the results of this process are outlined and motivated, before concrete objectives based on said results are defined.

Given the perpetual struggle of cloud platforms to keep up with industrial demand for supporting ever more demanding and diverse workloads outlined in §1.1, computationally challenging workloads that are not considered typical cloud applications are at first glance an excellent target for this work. Among these, high-performance computing (HPC) applications are particularly interesting, since moving such workloads to the cloud is a relatively novel trend that may yield massive cost savings and flexibility benefits compared to hosting the necessary infrastructure locally [18, 19, 20, 21]. Moreover, since performance is by definition a key requirement for HPC workloads, optimizations focussed on such workloads are highly relevant to practitioners.

While HPC workloads are highly varied in nature, one characteristic they all have in common is their emphasis on parallelism [22, 23]. This concept is

often implemented at two levels within these applications: shared-memory parallelism on the one hand, and distributed-memory parallelism on the other [24, 23]. The former is also known as multithreading and encompasses multiple application stacks executing in parallel within the same memory space. The latter involves multiple distinct processes—often hosted on distinct physical systems—cooperating through some communication protocol. While both of these concepts exist outside of the context of HPC, multithreading is much more commonly employed than distributed memory parallelism, with applications in web servers, data bases, video games, etc. Moreover, the low-level mechanisms employed by distributed memory parallelism are more numerous and vary greatly between applications, which severely limits the real-world impact of improving any specific cog in the distributed memory mechanism. For these reasons, the scope of this work is limited to multithreading.

Limiting the scope in terms of workloads to multithreaded applications also greatly reduces the variety of virtualization technologies to be considered. Namely, since multithreading is a purely computational concept, only central processing unit (CPU) virtualization is relevant to this work. Furthermore, the vast majority of cloud infrastructures are built around a single CPU architecture, namely x86 [25]. Finally, while many virtualization techniques exist for this architecture [26], hardware-assisted virtualization is by far the most popular technique these days [27, 28]. Combining all of the above, the scope of this work is limited to hardware-assisted virtualization of multithreaded workloads on x86 CPUs.

Combining the problems described in §1.2 with the scope constraints outlined above yields the main research question to be answered in this dissertation:

How can the performance cost of hardware-assisted virtualization of multithreaded applications be further reduced on the x86 platform?

This question implies addressing both problems listed in §1.2. However, it is evident that both of these problems cannot be resolved simultaneously. Concretely, the state of the art must be known before solutions to remaining challenges may be devised. Therefore, the first major contribution of this work is clarifying the state of the art regarding hardware-assisted virtualization of multithreaded applications on the x86 platform. This includes both assessing the effectiveness of the latest enhancements to the relevant technologies and identifying remaining challenges. Thus, the following secondary research questions shape the first stage of this dissertation:

- *What causes high hardware-assisted virtualization cost for multithreaded applications on the x86 platform?*
- *How effective are existing hardware-assisted x86 virtualization techniques at addressing the issues arising from virtualizing multithreaded applications?*

Once the remaining challenges regarding hardware-assisted virtualization of multithreading on the x86 platform are known, novel solutions may be devised to address said challenges. Besides merely describing such solutions, evidence for their effectiveness should be provided. This yields the second pair of secondary research questions to be addressed:

- *Which techniques can reduce the cost of hardware-assisted virtualization of multithreaded applications on the x86 platform?*
- *How can evidence for the efficacy of proposed techniques to reduce the cost of hardware-assisted virtualization of multithreaded applications on the x86 platform be provided?*

Providing evidence for the efficacy of the proposed techniques implies performing a comprehensive performance analysis. Empirical methods are to be preferred for this because of the complexity of virtualized systems. Moreover, performing an empirical performance analysis implies implementing the devised solutions, which allows them to be readily adopted by practitioners. This ensures that aside from scientific contributions, this dissertation has the potential to directly ameliorate industrial practices. This fits perfectly within the profile of the faculty of engineering technology, at which the Ph. D. project presented in this thesis has been conducted.

1.4 Synopsis

In order to answer the research questions outlined above, a thorough understanding of virtualization is required. Therefore, chapter 2 provides a comprehensive introduction to this concept. This chapter will make clear that virtualization is a complex process with many incarnations. Evidently, this makes measuring the performance cost of virtualization, i.e. virtualization overhead, a complicated task. Chapter 3 elaborates on how this virtualization overhead may be defined and measured and lists the principal known causes of virtualization overhead. Chapter 4 proceeds to address the first pair of partial research questions outlined in §1.3 by applying the techniques established in

chapter 3. A reflection on existing and potential future techniques to address the issues discovered in chapter 4 follows in chapter 5, providing an answer to the third partial research question formulated in §1.3. Finally, the last partial research question is addressed by implementing some of the techniques proposed in chapter 5 and empirically determining their effectiveness. Three complementary solutions have been selected for this detailed analysis: one at system level, one at application runtime environment level and one at application level. Chapters 6, 7 and 8 are each respectively dedicated to one of these. They each provide a deep dive into the problem they address, discuss the implementation of the proposed solution in both abstract and concrete terms and conclude with empirical evidence for the latter's efficacy. Finally, chapter 9 formulates a general conclusion.

Chapters 3 to 8 are all based on peer reviewed and published original work by the author of this dissertation, his colleagues and supervisors. Each of these chapters starts with a full bibliographic reference to the publication on which it is based and concludes with an outline of the main author's personal contributions to the work. Concretely, the following publications have been incorporated into this dissertation:

- **Chapters 3, 4 and 5**: S. Schildermans et al. "Virtualization Overhead of Multithreading in X86 State-of-the-Art & Remaining Challenges". In: *IEEE Transactions on Parallel and Distributed Systems* 32.10 (2021), pp. 2557–2570;
- **Chapter 6**: S. Schildermans et al. "Paratick: Reducing Timer Overhead in Virtual Machines". In: *50th International Conference on Parallel Processing*. 2021, pp. 1–10;
- **Chapter 7**: S. Schildermans et al. "Ptlbmalloc2: Reducing TLB Shootdowns with High Memory Efficiency". In: *ISPA-BDCloud-SocialCom-SustainCom 2020* (2020), pp. 76–83;
- **Chapter 8**: S. Schildermans and K. Aerts. "Towards High-Level Software Approaches to Reduce Virtualization Overhead for Parallel Applications". In: *2018 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE. 2018, pp. 193–197.

Chapter 2

Background: Virtualization

Central to this work is the concept of virtualization. This chapter clarifies this broad topic through describing all of the major forms of virtualization commonly utilized today. Even though the scope of this work is limited to a handful of specific aspects of virtualization, discussing the wide landscape of virtualization technologies as a whole allows for proper positioning of this work within the state of the art and eases interpretation of the presented findings.

2.1 Definition

Through the years, virtualization has broadened in scope to such an extent that it has become difficult to define unambiguously. Consequently, several accepted definitions exist today. One of the most prominent among these, to which this dissertation adheres, is the following [33]:

Virtualization is a technology that combines or divides computing resources to present one or many operating environments using methodologies like hardware and software partitioning or aggregation, partial or complete machine simulation, emulation, time-sharing, and many others.

According to the above definition, the principal purpose of virtualization is separating the notion of the operating environment from that of its underlying resources. Both the resources being virtualized and the technologies employed in doing so may vary greatly. Interestingly, aggregation and partitioning are both mentioned as forms of virtualization, even though these techniques achieve

opposite goals. The mainstream notion of virtualization is by contrast limited to partitioning alone.

While the above definition exemplifies that the applications and implementations of virtualization are highly diverse, all forms of this technique share the following conceptual structure:

- **Host**: The resources hosting the virtualized environment. Depending on the virtualization technique being applied, these resources may or may not be aware that they are being virtualized and may or may not provide specific support for this process. Often, these resources are selected to be similar to the virtual resources being created for performance reasons;
- **Virtualization layer**: A dedicated software component responsible for mapping requests to a virtual resource onto available physical resources. While the implementation of this layer may vary greatly, it always has the following properties [6]:
  - **Accuracy**: The virtualization layer must create an operating environment that accurately mimics the resources it represents. Note that these resources do not necessarily need to exist physically. A virtualization layer may for example create an operating environment representing a CPU employing an instruction set architecture (ISA) not used by any real CPU. However, it must accurately represent this fictitious CPU such that the virtual operating environment behaves in a predictable, well-defined and correct manner;
  - **Efficiency**: The virtualization process must not be excessively costly in terms of resource consumption or performance. Because some resources may be much more difficult to virtualize than others, 'excessively costly' is not defined in concrete terms. Nevertheless, efficiency must be a key design goal of any virtualization technology;
  - **Hardware control**: The virtualization layer must have full control over the resources being virtualized, such that it is impossible to change the state of the physical system from within the virtual operating environment in a problematic way without the virtualization layer being able to intervene.
- **Virtual machine**: The environment the virtualization layer creates. Two types of virtual machines (VMs) exist: process VMs on the one hand, and system VMs on the other. The former represent a virtualized environment for a single process, while the latter are virtualized representations of entire systems, in which multiple processes may be hosted [34]. All VMs share the following properties, which naturally follow from the properties of the virtualization layer [6]:

  - **Efficiency**: VMs must exhibit comparable performance to their physical counterparts;
  - **Isolation**: VMs must be strictly isolated from one another, as well as from the host system (unless they are explicitly configured otherwise);
  - **Accuracy**: VMs must from the perspective of the entities consuming them accurately represent the resources they mimic.
- **Guest**: The entity consuming the virtualized resources.

Beyond the above, little can be said about virtualization as a whole, again due to the breadth of the field. Nevertheless, a deeper understanding of the intricacies of various virtualization technologies is evidently paramount in the context of this dissertation. While discussing each of these technologies separately would be prohibitively onerous, many related technologies can be grouped, effectively splitting the virtualization spectrum in distinct categories. Kampert et. al. provide such a categorization, based on the type of resource being virtualized [17]:

- **Hardware virtualization**: Virtualizing the hardware with respect to the OS;
- **Operating system virtualization**: Virtualizing the operating system (OS) with respect to applications;
- **Application virtualization**: Virtualizing the system with respect to a single application;
- **Desktop virtualization**: Virtualizing the desktop environment with respect to end users;
- **Storage virtualization**: Virtualizing storage with respect to the OS or applications;
- **Network virtualization**: Virtualizing the network with respect to the OS or applications.

Note that many other categorizations of virtualization technologies may be devised. Moreover, some technologies may not easily fit within a single category. For example, one may argue that networking and storage are both supported by physical devices and are therefore forms of hardware virtualization. However, one may equally argue that 'storage' and 'networking' are high-level concepts entirely separate from their physical implementation and therefore require dedicated categories. After all, one may perfectly grasp the idea of a 'computer

network' without having any idea of how such a network would be implemented. This work opts for a middle ground between these views by including the low-level technicalities of virtualizing I/O devices in the category of hardware virtualization, while retaining dedicated categories for discussing the high-level concepts of virtualized storage and networking. The remainder of this chapter elaborates on each of the above categories, emphasizing those most important to this dissertation.

2.2 Hardware Virtualization

Hardware virtualization -more specifically hardware partitioning- is what laymen most often refer to with the term 'virtualization'. This is the most fundamental form of virtualization, as it virtualizes physical hardware with respect to the OS [33]. The virtualization layer is in this case a stand-alone, kernel-like software program referred to as the hypervisor or virtual machine monitor (VMM). VMs created through this virtualization method are always system VMs.

Any modern general purpose computer system is comprised of a variety of different hardware components, each serving a distinct purposes and as such exhibiting a distinct architecture and behavior. Therefore, virtualizing each of these components also requires distinct techniques. Much like with virtualization as a whole, in literature similar hardware components are often grouped together and discussed as a whole, since elaborating on the specifics of each component is infeasible. Most often, the categories distinguished in literature are CPU virtualization, random access memory (RAM) virtualization and I/O virtualization. This section elaborates on each of these categories below. Firstly however, the VMM is discussed in depth, as it is central to all three of these categories.

2.2.1 The Virtual Machine Monitor

Of all virtualization categories identified in §2.1, hardware virtualization requires the most complicated virtualization layer. The reason for this is that for this form of virtualization, the intended guests are most often fully-featured operating systems, which assume to be in direct control of the hardware. They will therefore often attempt to perform operations which are perfectly safe in a bare metal context but problematic in a virtualized one, where resources must be shared with other VMs and the host system alike. Examples include allocating memory, accessing I/O devices, etc. The VMM must identify whenever a guest attempts to perform such a sensitive operation and replace that operation with

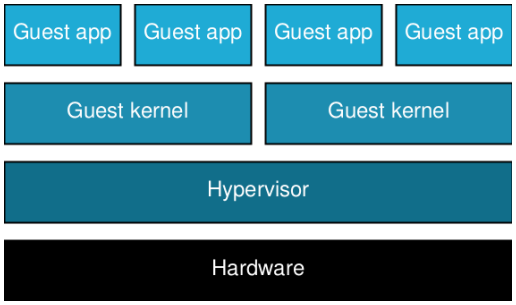


Figure 2.1: Schematic overview of a system stack employing a type 1 hypervisor.

a(n) (sequence of) operation(s) which emulate(s) it without compromising the system. In the interest of efficiency, other guest operations are to be executed directly on the hardware to the extent possible [6, 35].

From the above, it is evident that the VMM in essence acts as an operating system for operating systems. Much like the OS is a layer between the hardware and applications, the VMM is a layer between the host system and guest kernels. Much like the OS provides a virtual operating environment to applications which grants them the illusion they have the entire system at their disposal, the VMM provides a VM to guest kernels which grants them the illusion they have the entire system at their disposal. Much like the OS multiplexes physical resources between applications, the VMM multiplexes physical resources between guest kernels. Much like the OS strictly separates applications and intercepts illegal application behavior, the VMM strictly separates guest kernels and intercepts any attempt of theirs to alter the system state in a manner which is not permissible. Knowing that historically operating systems were refered to as 'supervisors', this analogy explains the term 'hypervisor', as a streamlined version of 'supervisor supervisor' [6, 35].

VMMs exist in various forms, all of which are commonly used today. Each of these forms is described in detail below.

Type 1 Hypervisors

The most commonly used VMMs are type 1 or bare metal hypervisors. These hypervisors run directly on the hardware and therefore have full control over it [36, 37]. Figure 2.1 schematically shows a virtualized system stack employing a type 1 hypervisor.

Type 1 hypervisors are highly popular in industry because they are only constrained by the hardware in performing their function. This brings several advantages:

- **Reliability**: Their design is relatively simple, which makes them robust.
- **Configurability**: They allow for pervasive system configuration through features such as live migration of VMs between physical systems, overallocation of resources, automatic snapshot creation, etc. All of this is usually configured remotely through dedicated management software.
- **Performance**: They have direct access to all hardware features, without having to pass through intermediate interfaces. This allows them to emulate problematic guest actions as efficiently as possible.

Many type 1 hypervisors exist, each with its own peculiarities. Their performance and capabilities are however very similar [7]. Examples include Xen[1], VMWare ESXi[2] and Microsoft Hyper-V[3].

Type 2 Hypervisors

Type 2 or hosted hypervisors are VMMs running as an application on top of a host OS [38, 36]. In contrast to type 1 hypervisors, the host may thus be utilized as a bare metal system in tandem with the virtualization infrastructure. Figure 2.2 illustrates such a system topology schematically.

The hosted nature of type 2 hypervisors limits them with regard to emulating problematic guest operations. Specifically, rather than directly manipulating the hardware, type 2 hypervisors must make do with the interfaces provided by the host OS. As a result, type 2 hypervisors offer fewer features and perform worse than their bare metal counterparts. Moreover, they are much more complex and therefore less robust than type 1 hypervisors [39]. All of these limitations result in type 2 hypervisors rarely being used in industry.

Despite the issues surrounding type 2 hypervisors, they offer the major advantage of flexibility. A type 2 hypervisor may be installed on any system without impacting other functions that system may be performing. Therefore, type 2 hypervisors are popular among amateur users who wish to e.g. run software not supported by their OS. The most popular type 2 hypervisor at the moment is Oracle VirtualBox[4].

[1] `https://xenproject.org/`
[2] `https://www.vmware.com/products/esxi-and-esx.html`
[3] `https://docs.microsoft.com/en-us/virtualization/hyper-v-on-windows/about/`
[4] `https://www.virtualbox.org/`

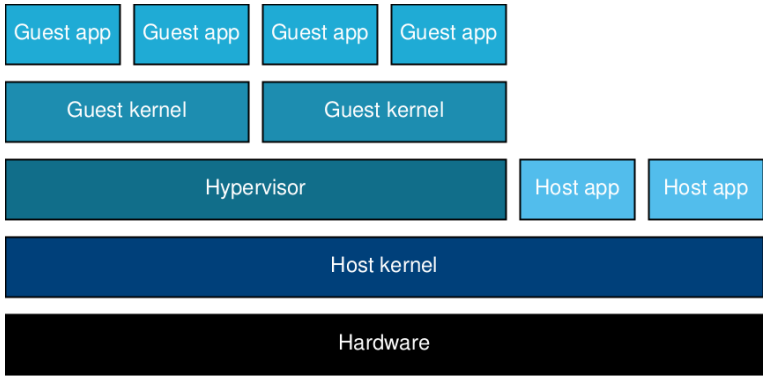


Figure 2.2: Schematic overview of a system stack employing a type 2 hypervisor.

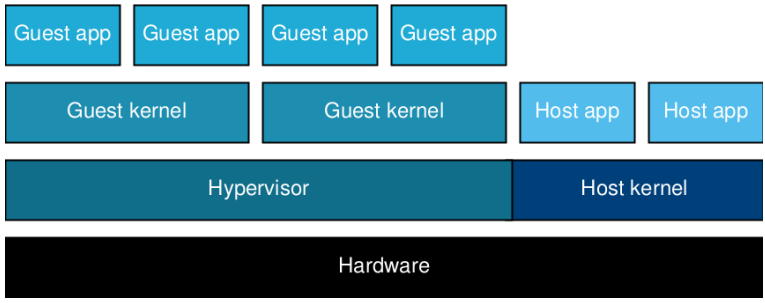


Figure 2.3: Schematic overview of a system stack employing a kernel-assisted hypervisor.

Kernel-Assisted Hypervisors

The final type of VMM is in essence a hybrid of type 1 and type 2 hypervisors. As the name implies, a kernel-assisted hypervisor is an integrated component of the host kernel [40]. This allows a general-purpose OS to provide virtualization, yielding a bare metal host system and a VMM with direct hardware access in one package. Figure 2.3 illustrates this concept.

Kernel-assisted VMMs combine the advantages and avoid the drawbacks of type 1 and type 2 hypervisors. This makes them attractive to both industrial and private users. However, being a relatively novel technology, their popularity is currently limited. Only one mainstream kernel-assisted VMM exists at the moment, namely KVM[5], which is implemented as a Linux kernel module.

[5] `https://www.linux-kvm.org/page/Main_Page`

2.2.2 CPU Virtualization

The CPU may be described as the heart of almost any computing system, as it is responsible for executing instruction streams that make up applications. Virtualizing the CPU encompasses abstracting these instruction streams from the physical CPU (pCPU). This is achieved through the concept of virtual CPUs (vCPUs). From the perspective of the host, these vCPUs may be seen as processes. From the perspective of the guest however, they are indistinguishable from pCPUs. Thus, the guest schedules its processes onto vCPUs, which the VMM schedules onto pCPUs as it sees fit [34].

As stated in §2.2.1, guest kernels will often perform operations that are not permissible in a virtualized context. Many of these operations manifest themselves as CPU instructions. Such instructions are called 'sensitive instructions'. Whenever a vCPU attempts to execute such an instruction, the VMM must be made aware thereof. To achieve this, early VMMs exploited the layered privilege model most CPUs possess. This model consists of at least two privilege levels: kernel mode (ring 0 in x86) and user mode (ring 3 in x86). Kernel mode is unrestricted and usually reserved for the kernel, while user mode only allows access to a subset of the ISA and can therefore safely be used by all applications [41, 42]. If a program executes a privileged instruction in user mode, the CPU passes control to the kernel, which may handle the incident as it sees fit. Therefore, as long as the set of sensitive instructions is a subset of the set of privileged instructions, registering the VMM as the 'kernel' and executing all vCPUs in user mode guarantees that the hardware will pass control to the VMM whenever a guest attempts to execute a sensitive instruction. The VMM may then emulate the sensitive instruction as it sees fit, after which it may resume vCPU execution. This process is called trap-and-emulate or classic virtualization [6].

Unfortunately, most modern CPU architectures, including ARM and x86, may not be virtualized through classic virtualization, as some of their sensitive instructions are not part of the set of privileged instructions. Therefore, virtualizing these architectures was long thought to be impossible [43, 33]. However, several methods have been devised through the years to work around this issue, allowing virtualization of these architectures after all. For x86 in particular, three such methods have been widely adopted. Each of these is described in detail below.

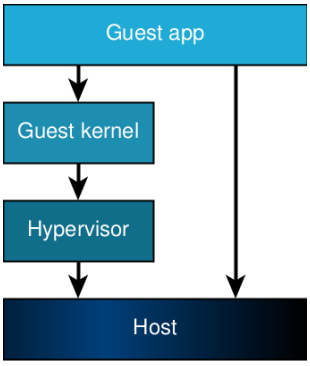


Figure 2.4: Schematic overview of dynamic binary translation.

Dynamic Binary Translation

Dynamic binary translation is based on the concept of emulation, which involves interpreting guest instructions one by one [44]. Although emulation is still widely used today due to its versatility, it can not be considered a form of virtualization because of its enormous performance cost [6]. After all, each guest instruction must be read and replaced by a (sequence of) host instruction(s) before being executed, which is even with the most up-to-date techniques a costly affair [45].

Dynamic binary translation sacrifices some of the emulation's flexibility in favor of performance. Specifically, in contrast to emulation, it assumes that the host and guest ISA are identical. This allows for direct execution of all user mode guest instructions. Only kernel mode instructions must be interpreted by the VMM [44, 26, 46]. Figure 2.4 shows this schematically.

Most VMMs based on dynamic binary translation employ a number of additional optimization techniques. For example, instructions may be translated in batches rather than one by one. Commonly recurring sequences of instructions may even by cached by the VMM [47].

The main advantage of dynamic binary translation is its versatility, since the VMM requires host nor guest support. Therefore, this technique is particularly suitable for certain niche applications such as nested virtualization [48]. The main drawbacks of this method are the complexity of the VMM and the relatively high performance cost this technique incurs despite all optimizations. Therefore, other virtualization techniques are preferable in most scenarios.

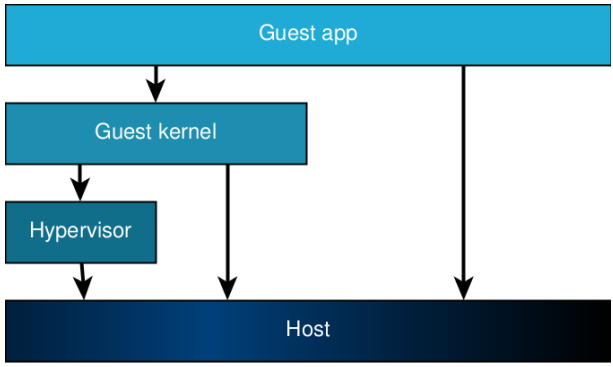


Figure 2.5: Schematic overview of paravirtualization.

Paravirtualization

The core properties of virtualization listed in §2.1 imply that guests must be unaware of the fact that they arre being virtualized. Forms of virtualization strictly adhering to this property are referred to as full virtualization [26]. However, it is self-evident that a guest who is aware of its virtualized status is able to proactively avoid executing sensitive instructions and request VMM intervention when needed, making the virtualization process much less complex and resource-hungry. This is exactly the core tenet of paravirtualization.

VMMs employing paravirtualization may be viewed as implementations of application binary interfaces (ABIs) that guest kernels may call when they need to perform a sensitive operation, much like applications may call upon the OS to perform privileged operations on their behalf through the system call interface. As such, this ABI is aptly named 'the hypercall interface'. Technically, trusted guest kernels may be executed directly on the hardware in kernel mode. However, for security reasons, limiting guests to user mode is still strongly recommended [37, 26, 33, 49]. All of this is shown in figure 2.5.

Paravirtualization is highly efficient since the guest OS and VMM actively cooperate, contrary to full virtualization. Moreover, VMMs for paravirtualization are relatively simple. The principal drawback of this technique is however that much like applications must be compiled and/or linked for the specific OS they target because the system call interface is OS-specific, paravirtualization requires guest kernels to be modified for each specific VMM they are to be hosted by because the hypercall interface is VMM-specific. This severely limits the flexibility of this technique.

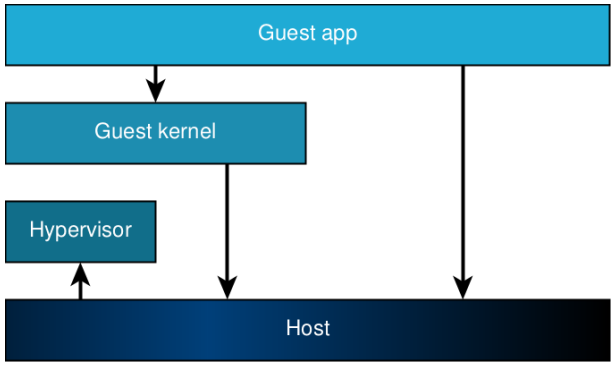


Figure 2.6: Schematic overview of hardware-assisted virtualization.

Hardware-Assisted Virtualization

In the early 2000's, both major x86 CPU manufacturers, Intel and AMD, released a new generation of processors which sported dedicated extensions to the ABI for virtualization. In essence, these extensions made the x86 architecture fully classically virtualizable. While implementation details differ, conceptually both manufacturer's technologies are mostly identical [50, 51]. Collectively they are known as hardware-assisted virtualization.

The main innovation behind hardware-assisted virtualization is the addition of an entirely new CPU operating mode, called 'non-root mode'. This mode is dedicated to running VMs. The traditional mode of CPU operation, which is used for all other software, has been renamed to 'root mode'. Both modes contain all four traditional privilege rings [27, 26, 37, 21, 46].

In non-root mode, guest code may safely run directly on the host, as shown in figure 2.6. The hardware keeps track of each guest's state using a dedicated data structure called the virtual machine control structure (VMCS). When a guest attempts to perform a sensitive operation, the hardware will autonomously switch to root mode and grant control to the VMM, saving the guest state in the VMCS. This is called a VM exit. The VMM may handle the VM exit as it sees fit, after which it may return control to the guest by performing a VM entry. The hardware reconstructs the guest state from the VMCS and resumes its execution.

Hardware-assisted virtualization achieves comparable performance to paravirtualization while still maintaining full virtualization. As such, it is the most popular virtualization technique today. Its only major drawback is its reliance on hardware support.

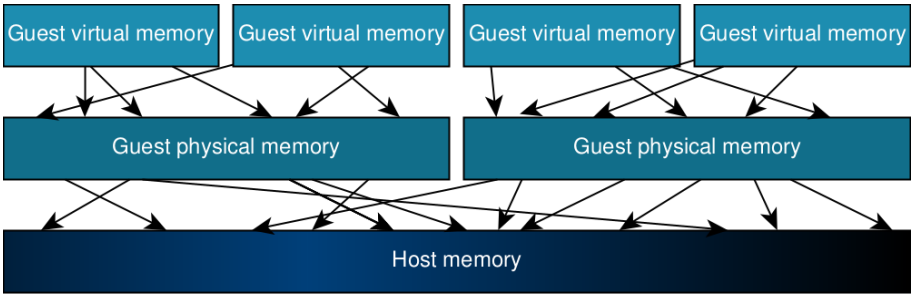


Figure 2.7: Schematic overview of multi-layered address translation.

2.2.3 Memory Virtualization

Much like the OS virtualizes memory with respect to applications, the VMM virtualizes memory with respect to guest kernels. The goal of this process is to present guests with a linear memory space sized in accordance with the amount of memory the guest kernel believes to have at its disposal, while the underlying physical memory may be sized and organized differently. This effectively introduces a double abstraction layer between guest application memory space and physical system memory, as illustrated in figure 2.7 [26, 37, 46].

Older CPU architectures tend to leave memory management entirely up to the OS. This makes addressing the double abstraction problem outlined above relatively simple, because the VMM only has to virtualize OS data structures. Contemporary architectures however, including x86, often contain advanced memory management units (MMUs) which integrate memory management with physical hardware. For example, x86 MMUs are able to perform page walks entirely in hardware. Retrieved PTEs are automatically stored in a dedicated cache, called the translation lookaside buffer (TLB). This means that virtualizing memory requires manipulation of the physical hardware as well as the OS data structures supporting it, which is evidently a complicated affair.

Three memory virtualization techniques are applicable to x86. All of these are still commonly used, as none is universally superior to the others [52, 53]. Below each of these methods is discussed in detail.

Shadow Paging

Shadow paging is the oldest memory virtualization technique. The guest operates as usual, maintaining page tables mapping guest-virtual addresses (GVAs) to guest-physical addresses (GPAs). For each of these page tables, the VMM maintains a shadow page table mapping GVAs to host-physical addresses (HPAs). The VMM marks the guest page tables as read-only, which allows it to intercept any guest page table modification. Upon each such modification, the VMM writes the GVA to GPA mapping to the guest page table, if necessary allocates new physical memory to the VM and adds the mapping from GVA to HPA to its shadow page table [54, 55].

For systems employing basic MMUs, the hardware will pass control to the VMM upon each TLB miss. The latter may handle the miss by traversing the shadow page table. This is however not possible for systems performing page walks and TLB management in hardware (e.g. those based on x86). Instead, the VMM must change the value of the page table base address register (`CR3` in x86) from the base of the guest page table to the base of the shadow page table upon each context switch. Thus, the hardware and guest OS use completely distinct page tables.

Shadow page tables achieve bare metal performance in handling TLB misses, since they bypass the GPA through directly mapping GVAs to HPAs. On the other hand, handling page faults is very costly. Upon each page fault, the CPU transfers control to the guest OS, which will attempt to install a new mapping in the page table. This generates a hardware exception and traps to the VMM, which installs the mapping in both the guest page table and the shadow page table before handing control back to the guest. Since memory intensive applications may generate large amounts of soft page faults while other applications may generate almost none, the performance cost of shadow page tables may vary from negligible to crippling.

Direct Paging

The concept of paravirtualization is not limited to the CPU. Direct paging or MMU paravirtualization refers to paravirtualizing main memory [56, 53]. While with shadow paging the VMM provides the hardware with the GVA to HPA mapping entirely transparently to the guest, direct paging requires active cooperation of the latter. Concretely, the guest page tables contain direct mappings from GVAs to HPAs. The guest performs hypercalls to the VMM whenever it wishes to modify these page tables. The VMM thus fully takes over memory management from the guest OS.

Analogously to CPU paravirtualization (see §2.2.2), direct paging may achieve near-native performance by sacrificing flexibility. While guest kernels must be modified to implement this technique, no superfluous data structures must be maintained and no memory management operations must be intercepted or emulated to provide the guest with the illusion it is in control of the hardware. Xen is currently the only popular VMM employing this technique [57].

Nested Paging

Nested paging, extended paging and hardware-assisted memory virtualization are all terms for memory virtualization implemented mostly in hardware [55, 42, 53]. This technique allows the guest OS to maintain page tables as it would natively, mapping GVAs to GPAs. Simultaneously, the VMM maintains a secondary page table, mapping GPAs to HPAs. The MMU is aware of both of these page tables, allowing it to perform nested page walks in hardware, finding the mappings between GVAs and HPAs and installing them in the TLB. Moreover, the guest may modify its own page table, which means it can handle soft page faults without VMM intervention, as long as the GPA being mapped into the guest page table is mapped to a HPA in the secondary page table. If not, control is passed to the VMM, which allocates more physical memory to the VM and creates the needed mapping in the secondary page table. This is called an extended page table (EPT) violation.

Because nested paging allows most page faults to be handled without VMM intervention, in many cases it achieves near-native performance while maintaining full virtualization. However, the main drawback of this technique is that handling TLB misses can be very costly. Namely, a page walk normally requires up to n memory accesses, with n the number of page table layers. However, because nested page tables do not directly map GVAs to HPAs, each guest PTE must be translated to a physical address through the secondary page table, which may take up to n memory accesses. This process has to be repeated up to n times. Adding the accesses to the guest page table itself as well as those required to translate the page table root register (which contains a GPA), this yields up to $(n + 2) \times n$ memory accesses. For x86, which employs page tables with up to four layers, this means that nested paging requires up to 24 memory accesses to handle a TLB miss, instead of four in a native context. Nevertheless, since modern TLBs cover a vast address range, this drawback rarely outweighs the advantages of this technique. Therefore, all modern x86 CPUs supporting hardware-assisted virtualization offer nested paging and almost all VMMs use it by default if possible.

2.2.4 I/O Virtualization

Of all system component categories, I/O is by far the most diverse. Its constituents range from mouse and keyboard over block devices and network adapters to graphics accelerators. All of these devices have different capabilities and needs and therefore communicate differently with the system. Despite this diversity however, all I/O devices and accompanying drivers rely on only a handful of kernel mechanisms to provide their functionality. Thus, while optimal performance would require the VMM to implement dedicated support for each individual device, tackling a handful of fundamental challenges allows for efficient virtualization of the vast majority of I/O devices [58, 59, 60]:

- I/O devices are often orders of magnitude slower than other system components, which means that it is common for a VM to try to access a device which will be in use for multiple more milliseconds by the host or another VM. The VMM must thus efficiently and securely multiplex access to I/O devices across VMs;
- Each VM associates each I/O device's registers with a specific address range, being port numbers in the case of direct I/O or memory addresses in the case of memory-mapped I/O (MMIO). Moreover, direct memory addressing (DMA), which is often used by high-throughput devices, requires the guest to assign a memory region to a device. In a virtualized scenario, likely none of these identifiers correspond to the physical address range the device is using. All device accesses by the system and memory accesses by the device must thus be rerouted;
- Devices are usually configured through dedicated configuration registers. Guests can not be allowed access to these registers, since that would give them full control over the device. The VMM must thus intercept accesses to these registers and emulate their effect with respect to the VM;
- Interrupts delivered by I/O devices must be routed to the correct VM. If the recipient VM is currently preempted, the interrupt must be acknowledged and injected into the VM at a later time.

Multiple techniques exist to address the issues outlined above. Below the most common ones are described in detail [61, 62].

Device Emulation

Since the inception of hardware virtualization, emulation has been the standard method to virtualize I/O. This technique involves the VMM maintaining a virtual representation of some I/O device in memory and presenting this memory region to the VM. The latter sees this region as a physical device and thus uses one of its device drivers to interact with it. Any reads from or writes to the virtual interface are intercepted by the VMM and translated to instructions that may be passed on to the physical I/O device backing the virtual interface through the VMM's device driver. Interrupts are intercepted by the VMM, which injects them into the appropriate VM. Note that the emulated device interface represents an existing (often generic) device so that the guest may interact with it using one of its existing device drivers. The physical device corresponding to the emulated interface does however not need to match it (e.g. an emulated block device in RAM).

Device emulation is often implemented as an integrated VMM component. Some modern VMMs however delegate this task to user space. In this case, a dedicated I/O emulator runs as a host application besides the VMs. This application performs the bulk of the emulation work, only relying on the VMM to provide the physical device driver and the necessary plumbing to connect all components. This approach has multiple advantages. Firstly, the I/O emulator is a distinct system component, which means it can be exchanged for another emulator. Moreover, multiple VMMs may use the same emulator. Secondly, delegating I/O to user space reduces the footprint of the VMM. Since the VMM operates in kernel space and has full control over the system, minimizing its size reduces the system's attack surface.

The main advantage of device emulation is its flexibility. The VMM may present a unified interface to VMs regardless of the underlying hardware. This however comes at a great cost in performance, since every interaction between VM and I/O device requires VMM involvement. Nevertheless, device emulation remains the go-to I/O virtualization technique for most VMMs. Regarding dedicated user-level I/O emulators, QEMU[6] is the most popular example. This emulator is used by e.g. KVM and VirtualBox [63].

[6] `https://www.qemu.org/`

Paravirtualization

Much like with CPU and memory virtualization, the vast majority of performance cost associated with virtualizing I/O devices may be avoided if the guest is aware it is being virtualized. Paravirtualizing I/O starts with defining a virtual device interface which does not correspond to any existing physical I/O device. This interface may be thought of as a device API. A dedicated driver is installed in the guest kernel which directly interacts with the virtual interface through hypercalls. The VMM translates these to physical device commands through its own physical device driver [64].

Because their interface is designed specifically with virtualization in mind, paravirtualized I/O devices are often highly efficient. For example, their API-like nature forces guest drivers to refrain from relying on the state of memory registers, which avoids the need for the VMM to translate guest I/O ports, MMIO or DMA addresses to their physical equivalents. Again analogously to CPU and memory paravirtualization, the main downside of this approach is that each guest must implement the paravirtualized I/O driver. However, because adding such drivers is no different from adding physical device drivers and therefore requires no extensive changes to the guest kernel, I/O is currently the most popular application domain for paravirtualization. It is offered by e.g. Xen and QEMU.

Device Passthrough

As outlined above, one of the main challenges when virtualizing I/O is multiplexing the physical device between VMs. In many cases however, devices are only used by a single VM. In such scenarios, full virtualization of the I/O device is not necessary. Instead, the device may be passed through directly to the VM in question, which has exclusive access to it and may interact with it using its own device driver. The VMM only provides minimal abstraction of the device in the form of address remapping, directly passing through privileged I/O instructions and interrupts, so long as they do not pose a threat to the rest of the system.

Device passthrough achieves near-native performance in most scenarios. The obvious drawback of this technique is that device multiplexing between VMs is not possible. Additionally, VMs may only be migrated between physical hosts sporting identical I/O devices. Anyhow, while not applicable in all situations, the utility of this technique is obvious. As such, most contemporary VMMs offer support for device passthrough.

Hardware-Assisted I/O Virtualization

Even the most advanced software techniques can not virtualize I/O in a way that is simultaneously efficient and flexible. This is only possible by direct hardware support. During the past decade, more and more devices have started to implement said support as demand for virtualized I/O devices has grown exponentially. Due to the variety of devices and methods to interact with them, this hardware support for I/O virtualization can best be viewed as a collection of independent technologies, which, when combined, make I/O virtualization with little to no software support possible [65, 66]. Below some prominent examples of such technologies are briefly discussed.

DMA Remapping Modern chipsets implement the mapping from guest DMA addresses to physical ones directly in hardware [65]. This greatly increases DMA throughput by relieving the VMM of this responsibility.

Interrupt Remapping Analogously to DMA, interrupts may be remapped by the hardware itself. They may be directed to a specific CPU and its attributes, such as vector, delivery mode, etc. may be altered. Moreover, the hardware may be instructed to change the physical destination of an interrupt dynamically whenever its logical destination (i.e. vCPU) is migrated between pCPUs [65].

Security Domains The VMM may set up security domains for each device, defining which VMs may access which devices. This policy will be enforced by hardware through generating an exception when a VM attempts to access a device for which it is not authorized [65].

Besides further reducing the responsibilities of the VMM and thereby improving performance, when combined with DMA and interrupt remapping this feature allows for device passthrough to be implemented entirely in hardware. This hardware-assisted passthrough provides genuine native performance.

Interrupt Posting Through interrupt posting, hardware interrupts intended for a particular VM may be delivered directly to that VM without VMM intervention [42, 65]. This feature requires the VMM to assign a dedicated posted-interrupt vector to each VM. The VMCS contains a field for this vector as well as a pointer to a posted-interupt descriptor. This data structure contains a flag for each supported interrupt vector. When a device assigned to a particular VM fires an interrupt, the hardware remaps this interrupt to the posted-interrupt vector and records the physical vector in the posted-interrupt descriptor. The

advanced programmable interrupt controller (APIC) then delivers the remapped interrupt to the target vCPU, which retrieves the physical interrupt vector from the posted-interrupt descriptor. If the target vCPU has been preempted when the interrupt arrives, the hardware records the interrupt in memory and delivers it as soon as the vCPU is rescheduled.

Interrupt posting allows for many interrupts to be delivered without any VMM involvement. Note however that for devices that are not assigned to particular VMs -i.e. devices frequently accessed by multiple VMs in an intertwined fashion- the hardware is unable to directly map device interrupts to the correct posted interrupt vector. Such interrupts are instead delivered to some CPU, which passes control to the VMM (as it always does upon receipt of an interrupt not corresponding to the posted-interrupt vector). The latter is then responsible for remapping the interrupt to the correct posted-interrupt vector and updating the corresponding posted-interrupt descriptor. Once this is done, the interrupt may be delivered to and processed by the correct vCPU without VMM intervention at the receiving end.

SR-IOV Peripheral component interconnect express (PCIe) is one of the main I/O buses used in personal computers and servers today. Its applications include graphics processing units (GPUs), network interface cards (NICs), etc. Because many of these components demand high throughput, efficient virtualization of the PCIe bus is paramount. To this end, the SR-IOV standard was developed [66, 67]. It splits the notion of PCIe devices into physical and virtual functions. A device usually has a single physical function, but up to 256 virtual ones. The main difference between these is that the physical function allows the device to be configured, while the virtual functions are limited to transferring data. Evidently, only the VMM has access to the physical function, while the VMs are presented with one or more virtual ones. All of this is mostly implemented in the hardware and firmware. Nevertheless, VMM support is required. Since this technique drastically reduces the performance cost of I/O virtualization with minimal software support, it is increasingly commonly applied.

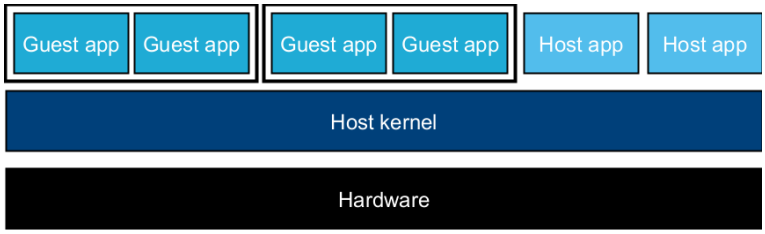


Figure 2.8: Schematic overview of OS virtualization.

2.3 Operating System Virtualization

Operating system virtualization, also known as containerization, is just like hardware virtualization a form of system virtualization. While both achieve similar effects, their implementations are entirely distinct. Where hardware virtualization virtualizes hardware with respect to the OS, OS virtualization virtualizes the OS with respect to applications [33, 68]. Concretely, the host kernel serves as the virtualization layer. All VMs share this kernel and may directly interact with it. These VMs—often called containers—perceive the kernel to be exclusive to them. Figure 2.8 illustrates this concept.

From a technical perspective, containerization is achieved by combining a number of capabilities naturally present in many OS kernels in such a way that multiple, fully isolated execution environments can be created within a single operating system context. Mainly, these capabilities are the following [33, 69]:

- **Dynamic file system root**: The file system is a core component of the environment the OS presents to applications. As such, each container must be assigned a unique file system root within the host file system. Moreover, the kernel must be able to dynamically switch between these file system roots when serving different containers;
- **Namespaces**: Namespaces are used to restrict resource access or avoid naming collisions in shared systems. Processes bound to a namespace do not have access to resources outside of that namespace. Furthermore, the scope of resource identifiers (hostnames, PIDs, etc.) is confined to namespaces. An identifier may thus point to distinct resources when defined in distinct namespaces;
- **Control groups**: When processes are competing for system resources, it is desirable to be able to monitor and manage resource consumption on a per-process basis. Control groups allow for exactly that.

Containerization achieves near-native performance because -as opposed to hardware virtualization- guest operations do not need to be intercepted or translated by the virtualization layer. Moreover, containers boot rapidly, since starting a container equates to initializing a process rather than booting a system. Lastly, containers require much less storage space and RAM than virtual machines due to the absence of guest kernels [70]. Countering these attractive benefits over hardware virtualization are inherently limited flexibility and security. Regarding the former, the fact that the kernel is shared between all containers implies that the entire system is limited to a single OS (family). Thus, while it is possible to virtualize different flavors of Linux using OS virtualization, creating a Windows container on a Linux host is out of the question. Regarding the latter, since containers share the host kernel and its interfaces (e.g. system calls), their attack surface is very large compared to that of classic VMs, which makes them unsuitable when security is a primary concern [71, 72]. While these drawbacks prevent it from fully replacing hardware virtualization, it has evolved to be a dominant force in the realm of system virtualization and by extension cloud computing.

Containerization has many flavors. In fact, most of the popular contemporary general purpose operating systems offer their own implementation of the concept. Broadly speaking, all of these flavors may be categorized into two groups: system containers and application containers. Below both are discussed in detail.

2.3.1 System Containers

Classic containerization partitions the host OS into fully independent system containers. Each container communicates directly with the kernel and behaves exactly like the OS it is based on, as shown in figure 2.9 [73, 74]. Resources are managed directly by the kernel on a per-container basis. Nevertheless, a minimal management daemon usually runs in the background to allow system administrators to easily configure and manage the containers.

Most UNIX-like systems sport their own version of system containers. Examples include Solaris Zones, FreeBSD Jails and LXC [75].

2.3.2 Application Containers

A recent development in containerization is the inception of application containers. While system containers are faithful OS replicas, application containers rather represent an application sandbox closely resembling an OS [76, 77]. From the sandboxed application's perspective, all regular OS interfaces

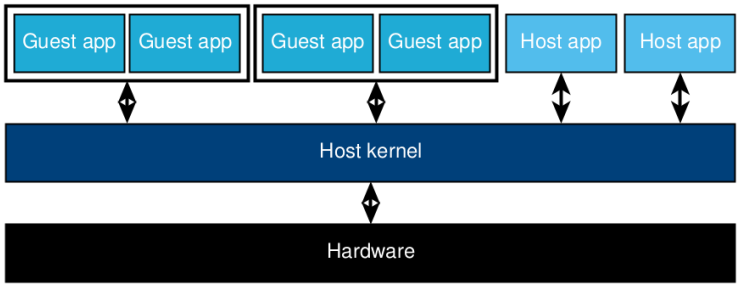


Figure 2.9: Schematic overview of system containerization.

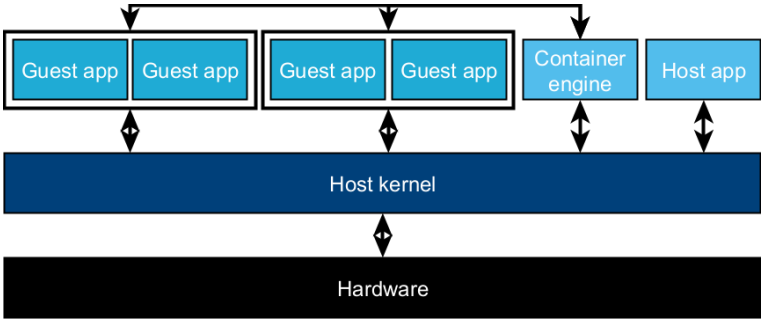


Figure 2.10: Schematic overview of application containerization.

are present. However, many resources are managed externally by a dedicated container engine. For these resources, the container engine acts as a broker between the host system and the container, as shown in figure 2.10. For example, rather than each container connecting directly to the host network, the container engine provides a virtual network to which the containers connect. Individual containers are not addressable from the outside world; only the container engine's virtual network adapter. Thus, while system containers resemble independent operating systems from the inside as well as the outside, application containers only do so from the inside. The main advantage of this approach is that the container engine may manage all containers running on the system as a group, which allows it to optimize resource usage. For example, containers are likely to share many system libraries. The container engine may share a single copy of these libraries between all containers, saving significant amounts of storage space.

While application containers are intended for use by a single application, there is no technical limit to the number of applications that may be hosted in such a container. It is however important to note that application containers are unsuitable for use as general purpose operating systems due to their reliance on external configuration. They are however a popular medium to distribute and deploy software, as they allow an application to be packaged with all its dependencies as a self-contained unit. Installing an application is therefore no more complicated than downloading and starting its container, which is trivial once the container engine has been set up. The most prominent example of application containerization is Docker.

2.4 Application Virtualization

Any technique to create process VMs (see §2.1) falls under the category of application virtualization. Conceptually, this category is much broader than the previously discussed ones. After all, computer programs are most often defined in terms of high-level logic, which must traverse several layers of abstraction before it may be executed on physical hardware. Strictly speaking, each of these abstraction layers can be viewed as a form of virtualization. Below the most important of these are described briefly.

2.4.1 Operating Systems

In the early days of computing, applications were written in machine language and executed directly on the hardware. Processes had to be loaded manually and could not run in parallel, which made systems inefficient and cumbersome [78]. To address these issues, the operating system was developed. Its fundamental task was -and is to this day- abstracting physical system resources from applications [79]. Each process is provided with its own virtual execution environment and is in principle unaware of any of the other processes on the system; having the illusion it has the entire system at its disposal. The OS provides a number of interfaces representing physical hardware functions, e.g. system calls and a virtual memory. Furthermore, it is responsible for multiplexing the physical hardware and enforcing isolation between processes. In this respect, the OS is the most fundamental form of application virtualization.

2.4.2 High-Level Programming Languages

Mostly for portability reasons, many high-level programming languages are not compiled directly to machine language. Instead, they are executed within a runtime environment, which translates application code to machine instructions. Many variations of this concept exist. For example, Python[7] programs are most often distributed as source code. At run time, a just-in-time (JIT) compiler performs minor optimizations before the code is fed to a platform-specific interpreter, which is responsible for converting this optimized python code to machine code and executing it [80]. On the other end of the spectrum, Java[8] programs are compiled in advance, albeit to highly optimized byte code rather than machine language. The Java runtime environment (JRE) transforms this byte code to machine instructions at run time. Thus, the compilation target of Java programs is the virtual ABI exposed by the JRE, also known as the Java virtual machine (JVM) [81].

2.4.3 Unikernels

Software architectures increasingly lean towards viewing applications as collections of loosely-coupled, autonomous services; as evidenced by the emergence of e.g. microservice architectures [82] and serverless computing [83]. These services are mostly passive entities, only performing work when an external request arrives. As such, they are expected to scale rapidly up and down in response to fluctuations in demand while still guaranteeing strict isolation between services. However, currently the unit of deployment in the cloud is a system VM containing countless interfaces, drivers, libraries etc., of which the majority is likely never used by a single service. Moreover, these system VMs provide process isolation at OS level, which is resource-hungry and entirely pointless in an environment where only one trusted service is being executed within an individual OS context. Unikernels have recently been introduced specifically to address these issues. They do this by asserting that a VM only has to statically support a single, predetermined service [84]. This allows for the kernel to be compiled, linked and executed as a cohesive unit with the application. This in turn erodes the notion of kernel and user space, which transforms system calls into simple function calls. Furthermore, since all application dependencies are known at kernel compile time, only kernel components needed by the application have to be provided. Moreover, these components can be highly optimized to suit that specific application.

[7] `https://www.python.org/`
[8] `https://www.java.com/en/`

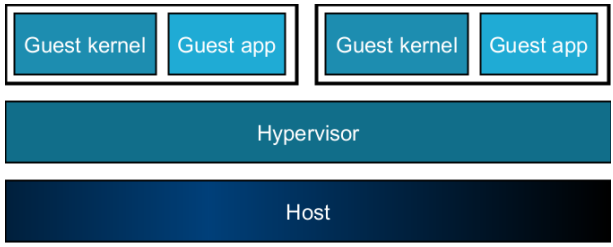


Figure 2.11: Schematic overview of a unikernel system.

Since unikernels do not provide resource management or process isolation, executing multiple applications requires installing multiple unikernels on top of a VMM. In fact, unikernels are intended to be used in this manner. This yields a collection of isolated, scalable, optimized VMs each supporting a single service: the ideal cloud infrastructure [85]. Figure 2.11 provides a schematic overview of all of this.

Because unikernels align exactly with the needs of the application they are hosting, their performance is excellent; often even exceeding that of bare metal general purpose OSs. Moreover, they boot much more rapidly than both classic VMs and containers, improving scalability. On top of this, compiled unikernel binaries are rarely more than a few megabytes (MB) in size [86, 87]. Thus, they offer performance exceeding that of containerization while maintaining the resource isolation offered by hardware virtualization. This emerging technology may therefore mature to one day dominate the cloud. In fact, some prominent members of the Linux community have already demonstrated a unikernel version of Linux, which they hope to integrate with the mainline kernel [88]. Examples of unikernels already available include OSv[9] and MirageOS[10].

A final note regarding unikernels is the recently proposed concept of unikernel monitors [89]. These can be viewed as VMMs integrated with the application, providing any needed virtual device interfaces. This approach relieves the host from much of the I/O virtualization work it should otherwise perform (see §2.2.4), further improving boot times and performance.

[9] `http://osv.io/`
[10] `https://mirage.io/`

2.5 Desktop Virtualization

In the 1970's, computers had become powerful enough to serve multiple users simultaneously. Combined with the fact that they were still too expensive to provide individual users with a personal computer, the mainframe concept was introduced. Multiple users could connect simultaneously to a powerful centralized computer through dumb terminals. These terminals provided a fully functional user interface, resembling a personal computer. This was the birth of the concept of desktop virtualization [26, 78].

Desktop virtualization is still used in large companies because it offers centralized resource management, as well as cost savings compared to purchasing personal machines for large numbers of employees. Another application of this concept is interfacing with embedded devices over a network [90]. Because this form of virtualization is not important to this dissertation, it is not further discussed.

2.6 Storage Virtualization

Perhaps the most commonly overlooked aspect of virtualization is storage virtualization. In fact, no modern storage device would be usable without some form of virtualization. Because storage virtualization is not a focus of this dissertation, this section is limited to a description of the most common variants of this technique, omitting technical details.

2.6.1 Logical Block Addressing

The most fundamental form of storage virtualization is logical block addressing (LBA) [91, 92]. Any modern storage device employs this technology. Namely, physical disks often use complicated optimization techniques such as parallel reading and writing (for performance) or spreading data evenly over the device (for longevity). Moreover, certain device areas may become defective over time. Exposing such details to the OS would be pointless and prohibitively complicated for the latter to handle. Therefore, storage devices are typically largely driven by integrated firmware, presenting a linear address space of usable storage blocks to the OS. The device itself translates requests to manipulate these blocks into physical hardware instructions.

2.6.2 Disk Partitioning

It is often desirable or even necessary for a storage device to be represented as multiple independent logical devices. For example, one may combine different file systems, restrict access to certain disk areas or provide redundancy. This may be achieved through disk partitioning. Usually, this technique is implemented by dedicating the first logical blocks of the disk to a partition table, which defines disk partitions and their properties. Each of these partitions is treated as an independent storage device by the OS [93].

2.6.3 Redundant Array of Independent Disks

Redundant array of independent disks (RAID) is a collection of techniques to aggregate multiple physical disks so that they appear as a single device to the OS [94, 95, 96]. It has many variants, the applications of which vary greatly. For example, RAID0 multiplexes all reads and writes between multiple disks in order to optimize performance. RAID1 on the other hand duplicates all writes across multiple disks in order to provide data redundancy. [97] provides a complete overview of the different forms of RAID.

2.6.4 Storage Area Network

Storage is traditionally viewed as an integral system component. Storage devices are therefore bound to the systems they belong to, and vice versa. This arrangement is referred to as directly attached storage (DAS). Storage area networks (SANs) on the other hand break this bond between system and storage [94, 95, 96]. A SAN consists of a centralized storage pool, managed by an appliance and presented as a singular block address space. Any number of systems may connect to the SAN and create partitions within the pool. Especially in large data centers this storage consolidation may lead to significant cost savings due to reduced fragmentation and simplified maintenance.

2.6.5 Network-Attached Storage

Network-attached storage (NAS) is conceptually comparable to SAN. The main difference between the techniques is their level of abstraction. While SAN aggregates disks at block level, NAS does so at file system level [95, 96]. Systems connected to a NAS network may thus mount ready-to-use file systems rather than having to allocate virtual disk partitions.

2.6.6 Software-Defined Storage

The most abstract form of storage virtualization is software-defined storage (SDS) [96, 98]. In essence, this technique is a refinement of SAN and NAS. Namely, both of these techniques are still constrained by the limitations of the proprietary hardware used in most of their implementations. SDS overcomes this by implementing the entire storage virtualization stack in software. This makes SDS cheaper and much more flexible at a minor performance penalty compared to traditional SAN and NAS.

2.7 Network Virtualization

Computer networks are essentially no more than physical connections between hosts to allow them to communicate. However, as the network grows, managing it efficiently requires specialized hardware, such as switches and routers. Because this hardware may be expensive and limited in flexibility, these days more and more of its functions are being implemented in software. This approach is often referred to as software-defined networking (SDN) [99]. Through SDN, system administrators may implement complicated network configurations using only basic hardware rather than expensive, specialized networking devices, albeit at a (limited) performance penalty. Below some of the most common applications of SDN are briefly described. Details are again omitted because this form of virtualization is not directly relevant to this dissertation.

2.7.1 Virtual Internet Protocol

In certain scenarios, the traditional network addressing technique of tying distinct internet protocol (IP) addresses to individual network interfaces is not desirable. For example, large-scale web services often can not be hosted on a single server for performance or reliability reasons. It would however be impractical to tie multiple addresses to a single service. This problem is addressed through the use of virtual internet protocol (VIP) [17, 100]. Usually, an IP address is assigned to a proxy server, which acts as the entry point to the web service. Clients connect to the proxy server, which then forwards their requests to any number of the back end servers, ideally spreading the load evenly.

2.7.2 Virtual Local Area Network

The internet (obviously the largest computer network in existence) is a two-tier entity: hosts within a single organizational unit such as a company, household, etc. are connected through a local area network (LAN). Individual LANs are in turn connected through the global wide area network (WAN) [101]. Traditionally, a router is the gateway between these two tiers of networking. However, often it is desirable to subdivide the LAN, mainly in the interest of security. Modern routers and switches therefore often allow for multiple virtual local area networks (VLANs) to be defined within a single LAN [17].

2.7.3 Virtual Private Network

One of the challenges arising from the two-tiered network topology described in §2.7.2 is that hosts belonging to different LANs can not communicate directly. While this is partly intentional for security reasons, it is not difficult to imagine scenarios where this limitation is problematic. For example, an employee may need to access data residing on a private company server while working from home. For this reason, virtual private network (VPN) technology was developed [102, 17]. In brief, VPN allows hosts to join a LAN over the WAN. This can be achieved in two ways:

- **Trusted VPN**: A direct physical connection between the external host and the LAN. This type of VPN is obviously very expensive and only used by major corporations to e.g. connect multiple office locations;
- **Secure VPN**: A VPN server is set up in the LAN and acts as a gateway to the WAN. External hosts may then connect to the VPN server through an encrypted tunnel. The latter then forwards connections between external and internal hosts.

2.8 Conclusion

Fundamentally, virtualization is no more than abstracting resources from entities aiming to employ those resources. As such, it is evident that this diverse technology is widely used in the information technology (IT) world, since abstraction is omnipresent in modern computing environments. In extreme cases, such as clouds, a multitude of virtualization technologies is combined to the extent that nor applications, nor system administrators, nor end users have any notion of the physical resources supporting the environment presented to them. For this reason, it is more appropriate to think of modern virtualized environments as interfaces rather than as resources. It is self-evident that continued optimization of this technology is instrumental as demand for flexible, affordable and efficient computing resources continues to surge.

Chapter 3

Virtualization Overhead

This chapter was previously published as part of:
S. Schildermans et al. "Virtualization Overhead of Multithreading in X86 State-of-the-Art & Remaining Challenges". In: *IEEE Transactions on Parallel and Distributed Systems* 32.10 (2021), pp. 2557–2570

Virtualization by definition introduces a layer of abstraction between operating environments and the resources supporting those environments, as outlined in §2.1. Since VMs equally by definition behave identically to their physical counterparts, this abstraction naturally causes some performance degradation which is referred to as 'virtualization overhead'. While at first glance this term is self-explanatory, defining and rigorously evaluating virtualization overhead are no trivial matters. To our knowledge, any existing work handling these topics employs its own ad-hoc definition of virtualization overhead—most often in terms of low-level performance metrics—and evaluates it through equally ad-hoc experiments. This obviously leaves much to be desired in terms of generalizability and correctness. This chapter addresses this lack of transparency through formally defining virtualization overhead and reflecting on the methodology most suitable to evaluate this overhead. Additionally, it lists the principal known underlying causes of virtualization overhead. Although the emphasis of this chapter lies on hardware-assisted virtualization of multithreaded applications on the x86 platform, many of the presented findings are generalizable to other scenarios and thus form a valuable and long-awaited contribution to the field.

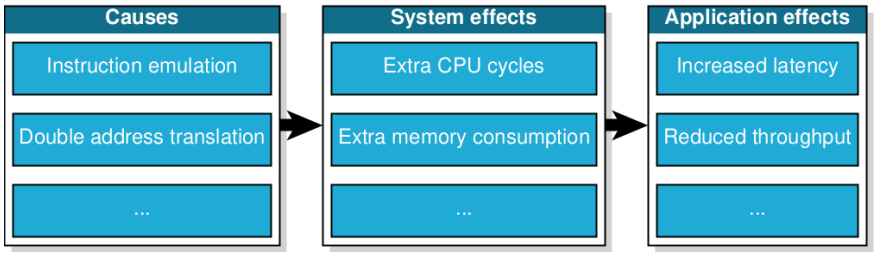


Figure 3.1: Schematic breakdown of virtualization overhead.

Main Findings & Contributions

- A definition for virtualization overhead that explicitly divides said overhead into internal system effects and external application effects is formulated;
- A general method for empirically evaluating virtualization overhead is described.

3.1 Definition

In order to deeply understand virtualization overhead, it is best approached from its root causes. These encompass any operation performed by the virtualization layer that intervenes with the normal execution of the VM. Examples include emulation of sensitive instructions, double memory address translation, etc. These operations all impact the system in some way, e.g. through requiring some CPU time to complete and consuming some memory. Finally, these system effects become visible to end users through negatively impacting application performance metrics such as execution time and application throughput. This causal relationship is crucial to an accurate definition of virtualization overhead. Figure 3.1 depicts it schematically.

In contrast to what intuition would suggest, system effects and application effects are not necessarily correlated. For example, when a server is not overloaded, I/O operations -which are notorious for inducing large amounts of virtualization overhead- may be offloaded to redundant CPU cores. In this way, the system effects induced by these operations are concealed from the guest and do not induce any application effects. This concealed cost can however not simply be ignored. Firstly, public cloud providers are charging consumers at ever higher resolutions to allow for fine-grained cost optimization [103]. For example, novel serverless cloud environments charge consumers per CPU-ms rather than

per VM-hour [83]. This means that consumed system resources are charged irrespective of their effect on the application. Secondly, concealed system effects may become visible to applications when the state of the system changes. For example, the offloaded I/O operations described above may start reducing application performance when the server experiences a sudden load spike which saturates all CPU cores. Thus, system and application effects need to be quantified independently, making virtualization overhead the sum of all the system and application effects a virtualized workload has on the system.

From the above, it is evident that both system and application effects must be understood thoroughly in order to understand virtualization overhead. The following subsections are dedicated to that purpose.

3.1.1 System Effects

Any excess internal system resource usage caused by virtualization (cycles, memory, bandwidth,...) is a system effect. However, within the context of this dissertation, only the system effects induced by virtualizing multithreaded applications are of concern. Since multithreading is a purely computational concept and the vast majority of its implementations target the CPU, CPU cycle consumption is naturally the main system effect of interest. While other metrics such as memory usage may be valuable as well, from a pragmatic perspective they only become important when they bottleneck the system. This will however be reflected by an increase in consumed CPU cycles. Knowing this, the system effects of virtualizing multithreading may be defined as follows:

Let $C_p(W, P(S_w))$ be the CPU cycles used by workload W on physical system $P(S_w)$, with S_w all settings for P. Let $C_v(W, V(S_w), P(S_v))$ be the system cycles used by W on a virtual machine $V(S_w)$ with the same settings, hosted on a system $P(S_v)$. Then the sum of all system effects is the reduction in resource efficiency induced by virtualization:

$$\delta\eta_r = \frac{C_v(W, V(S_w), P(S_v)) - C_p(W, P(S_w))}{C_p(W, P(S_w))}$$

In the above definition, S_v includes all system settings only visible to the VMM, e.g. the VMM used, concurrently running VMs, etc. S_w reflects all settings observable within the guest, e.g. concurrently running applications, vCPU count, etc. Note that these settings include both system configuration and the system state during workload execution.

3.1.2 Application Effects

Application effects are all effects induced by the virtualization process which are measurable externally and as such visible to end users through altering application behavior. Analogously to system effects, they encompass a variety of metrics such as latency, throughput, etc. Again analogously to system effects however, in the context of multithreaded, computation-intensive applications, any effects of interest from a pragmatic perspective may be translated to a single metric, being wall clock execution time. For example, reduced application throughput translates to either less work being done in the same time frame or more time being needed for the same amount of work. Thus, analogously to system effects, the sum of all application effects may be described as a reduction in temporal efficiency, $\delta\eta_t$, which is the increase in wall clock time needed to execute a workload W in a VM relative to the time needed by a physical system. One addition must be made though. Since wall clock time is measured externally and the system settings S_v may include temporally multiplexing physical resources between $V(S_w)$ and other tasks, the effective resources available to the VM must be taken into account. In other words, the effects of resource sharing must be separated from those of virtualization. Based on §3.1.1, the amount of available CPU time may be used as a proxy for system resources in the context of this dissertation. This yields the following definition for $\delta\eta_t$:

$$\delta\eta_t = \frac{t_v(W, V(S_w), P(S_v)) \times \gamma_v - t_p(W, P(S_w)) \times \gamma_p}{t_p(W, P(S_w)) \times \gamma_p}$$

With t_p and t_v the wall clock execution times for W in respectively the physical and virtual environments and γ_p and γ_v the ideal effective CPU count available to the workload given S_w and S_v in each respective environment. Note that γ disregards any system-level overheads and is based on the resources available to the workload and not the amount of resources the workload effectively uses. Thus, when a sequential application utilizing a single CPU is executed in an environment offering four CPUs, γ equals four.

3.2 Causes

From the previous section it is clear that virtualization overhead is merely a symptom of a variety of underlying issues. Many of these issues are by now well understood, since finding and mitigating the root causes of virtualization overhead has been the subject of countless scientific publications [104, 46, 105, 66, 106, 107, 14, 108, 109]. This section elaborates on the most important of these causes and describes any existing techniques to address them which are already widely adopted in industry.

3.2.1 Unfair Resource Allocation

One of the main purposes of virtualization is hardware consolidation. As a result, multiple VMs often share hardware resources. Due to inefficient resource management policies in the VMM or unmanaged contention between VMs, applications may be unnecessarily starved of resources such as CPU, cache or memory. Many efforts have been made to minimize this issue. Examples include memory deduplication [104] and Intel resource director technology (RDT) [42].

3.2.2 Instruction Emulation

At the VMM level, emulation of sensitive operations is still a major cause of performance degradation for certain workloads. While some virtualization techniques (i.e. paravirtualization) avoid this cost, doing so has other drawbacks such as reduced flexibility [46].

3.2.3 Input/Output

I/O operations, such as accessing I/O ports, DMA and interrupts all require special attention in a virtualized environment, as described in §2.2.4. Additionally, for high bandwidth I/O devices, extra data needs to be copied to the VMM. Techniques for working around these limitations include paravirtualization (e.g. paravirtualized drivers sharing I/O buffers between VM and VMM) [105] and hardware assistance [66].

3.2.4 Double Memory Address Translation

As described in §2.2.3, guest memory accesses have to be translated to VMM-managed machine addresses in virtualized systems. All existing techniques to implement this double address translation have some significant drawback: shadow page tables require VMM intervention upon each page fault (see §2.2.3), direct paging sacrifices flexibility (see §2.2.3) and nested paging causes page walks to be much more costly than in a bare metal setting (see §2.2.3).

3.2.5 Spinning Synchronization

Spinning synchronization involves a shared data structure called a 'spin lock' which may only be atomically read and updated. If a spin lock is free, a thread may claim it by marking it as claimed through an atomic operation. Any other threads attempting to claim the lock before the original thread has released it will continually poll it in a loop until it becomes available. A thread may free the spin lock by simply marking it as such through a regular write operation.

Because of their simplicity and minimal latency, spin locks are the preferred form of synchronization for short critical sections, especially when performance is of greater concern than efficiency. As such, spinning synchronization is often used within OS kernels. While in a native environment this is perfectly sensible, in a virtualized context spinning synchronization may be problematic. Namely, when the hardware is overcommitted, the VMM may deschedule a vCPU holding a spin lock in order to schedule a vCPU belonging to another VM, causing the vCPUs waiting for that lock to waste cycles. This is known as lock holder preemption (LHP) [110].

Many systems offer a more advanced version of spinning synchronization in the form of ticket spin locks [111]. A ticket spin lock may be viewed as a regular spin lock which additionally ensures that a contented lock is passed from thread to thread in the order in which the threads attempted to claim the lock. In this way, threads waiting for the lock are ordered in a first-in-first-out (FIFO) queue, which prevents thread starvation. In a virtualized environment, such primitives are even more problematic than regular spin locks because besides suffering from regular LHP, these locks may also cause excessive spinning when a spinning vCPU at the head of the wait queue is preempted by the VMM when the lock is released. In such a scenario, vCPUs behind said preempted vCPU are forced to spin for a prolonged period of time, despite the ticket spin lock technically being available. This problem is known as 'lock waiter preemption (LWP)' [109].

Several approaches have been proposed to mitigate the issues described above. Hardware extensions that trigger a VM exit when a vCPU executes excessive amounts of `PAUSE` instructions—indicating spinning—are already widely adopted. Intel's variant of this technique is called 'pause loop exiting (PLE)' [42] and AMD's is known as 'pause filter (PF)' [112]). Additionally, paravirtualized ticket spin locks largely mitigate LWP in Linux for the KVM and Xen VMMs [113]. Such locks operate like traditional spin locks by default (i.e. 'fast path') but switch to 'slow path' mode as soon as any vCPU has been spinning for a predetermined amount of time in an attempt to acquire the lock. Slow path mode entails that the spinning vCPU enters a blocking state and waits for the lock to become available (as do all vCPUs attempting to acquire the lock as long as it is in slow path mode). When a vCPU releases a paravirtualized ticket spin lock in slow path mode, it performs a hypercall to inform the VMM that the first blocked waiting vCPU in line may be rescheduled. If there are no other vCPUs contending for the lock at that time, it switches back to fast path operation [113, 114]. Paravirtualized ticket spin locks may thus be seen as a hybrid between spinning and blocking synchronization.

3.2.6 Blocking Synchronization

Blocking synchronization is a more efficient alternative to spinning synchronization because contented locks are not continually polled. The basis of this mechanism is analogously to spinning synchronization a simple shared data structure which may be claimed by atomically reading and updating it. In contrast to spinning synchronization however, threads attempting to claim the lock when it is not available enter a blocking state, yielding the CPU they were occupying to the OS. The latter may then schedule other tasks on this CPU, if any are available. If not, it issues a `HLT` instruction to put the CPU in a low power mode, saving energy. When the contended lock is released, the OS marks the blocked thread as runnable, so that it may be scheduled and claim the lock. If any idle CPUs are available, the kernel wakes one of them by means of a `RESCHEDULE` inter-processor interrupt (IPI), which invokes the scheduler on that CPU and allows the newly awoken thread to claim the lock and continue work immediately [115, 42].

Blocking synchronization is much more commonly used by applications than spinning synchronization due to its greater resource efficiency, especially for longer critical sections. However, while the blocking synchronization process may be highly efficient in a bare metal environment, in a virtualized context several complications arise:

- When a vCPU encounters a `HLT` instruction a VM exit is triggered, after which the VMM scheduler runs in order to find any other tasks to be executed on the corresponding pCPU. For relatively short critical sections this may prove problematic, since this process may take much longer than the time the blocking thread would have had to wait for the lock to become available. Because of this, most VMMs implement an optimization called 'halt polling' [15]. This involves the host first busy-waiting for a dynamically determined amount of time before scheduling out the vCPU. If the vCPU receives new work from the guest kernel during this time, the host resumes its execution immediately rather than scheduling a new task;

- Similarly to LHP, in an overcommitted setting the vCPU holding a blocking-based lock may have been preempted, which may cause many vCPUs to pointlessly block before the vCPU holding the lock is finally rescheduled and the application may make progress. This, in combination with the above issue, is known as the 'blocked waiter wakeup (BWW)' problem [14];

- For modern Intel x86 CPUs utilizing X2APIC, sending IPIs requires writing to the interrupt command register (ICR), which is a model-specific register (MSR) containing among other things the destination CPU of the IPI to be sent. Because in a virtualized environment the destination vCPU visible to the guest kernel may not correspond to the pCPU visible to the hardware, the VMM must intercept all ICR writes through a VM exit in order to remap the destination CPU address. On older systems not sporting interrupt posting (see §2.2.4), a second VM exit is required to deliver the IPI on the receiving CPU. When the guest is not under heavy load, it is likely that upon each release of a contended lock a `RESCHEDULE` IPI is sent to schedule the newly awoken thread on an idle vCPU, thereby invoking at least one VM exit.

3.2.7 Memory Consistency

An often-overlooked aspect of multithreading—especially in a virtualized context—is the effect of sharing data between threads executing concurrently on distinct CPUs. Namely, in X86, TLBs are almost always CPU-local and populated by hardware but—in contrast to other caches—synchronized by the OS [42, 115]. Because of the semantic gap between the hardware and the OS, the contents of each TLB are opaque to the latter. This means that whenever a CPU alters a PTE, the OS must notify all CPUs sharing the virtual address space to flush the altered PTE from their TLB. This is achieved by sending

IPIs to the CPUs meeting the condition just described and waiting for all of them to acknowledge the flush request before proceeding. This process is called a 'TLB shootdown'.

In a native setting, TLB shootdowns are generally considered sufficiently efficient. However, this mechanism has been shown to have problematic performance implications for specific workloads [116]. Adding virtualization to the equation exacerbates this issue in several ways:

- Much like the `RESCHEDULE` IPIs discussed in §3.2.6, sending a TLB shootdown IPI requires a write to the ICR MSR. A TLB shootdown thus invokes a number of VM exits proportional to the number of concurrently executing threads at the moment the shootdown is triggered;
- Because the vCPU sending a TLB shootdown must synchronize with all receiving vCPUs, which is most often implemented by means of a spin lock, a LHP-like problem may occur when one or more of the receiving vCPU(s) has/have been preempted by the VMM (see §3.2.5). This issue is known as 'TLB shootdown preemption' [117].

3.2.8 Non-Uniform Memory Access Opacity

Usually the guest is unaware of the exact physical hardware configuration. This can decrease e.g. cache and memory performance. Particularly for NUMA systems this is an issue, since NUMA-unaware scheduling can greatly increase memory and synchronization latencies [118]. Several solutions to this problem have been developed, such as NUMA-aware VMM schedulers [115], dedicated VMM-level NUMA locality managers [108] and exposing the NUMA architecture to the guest [119].

3.3 Quantification

Quantifying virtualization overhead is much like defining it not trivial. The complexity of modern systems makes empirical evaluation based on controlled experiments the only feasible approach. Analytical methods or simulations are likely less accurate and far more time consuming. However, designing meaningful experiments to evaluate virtualization overhead is challenging. Most problematic is the vast quantity of possible system settings that may drastically influence overhead for any workload. Moreover, when one aims to assess certain application properties rather than particular workloads—as is the case for this

dissertation—choosing a representative (set of) workload(s) in itself is no easy task. Finally, even with correct system settings and representative workloads, it is important to keep several best practices in mind when collecting data and transforming it into interpretable results. This section elaborates on all of these considerations and provides a template for experiments to evaluate virtualization overhead. Unless stated otherwise, all experiments presented in this dissertation conform to this template.

3.3.1 System Settings

At the heart of any computing system lies the hardware. Within the context of this dissertation, the CPU is by far the most important hardware component to consider, as it is central to both multithreading and hardware-assisted virtualization (see §1.3). Since Intel dominates the corporate x86 server CPU market, with AMD having a market share of only 8%, Intel-based systems are preferable [120]. However, results can be safely generalized to AMD-based systems, since Intel VT-x and AMD-V are nearly identical [42, 112]. To the best knowledge of the author of this dissertation, no studies suggest a notable performance difference in any regard between these technologies.

Because the hardware configuration influences performance as well as overhead, the number of CPUs effectively available to the workload under evaluation should be varied over a sufficiently wide range. The same applies to the number of NUMA sockets over which these CPUs are distributed. In a virtualized environment, this may be achieved by creating VMs with the appropriate vCPU counts and pinning those vCPUs to the appropriate pCPUs. In a native environment, scheduling tools such as `taskset`[1] may be used to pin processes to a set of CPUs. Since memory and I/O are no primary concerns for this dissertation, both are provided in abundance so as to minimize the chance they might form a system bottleneck.

Concerning hypervisors, four players dominate with a combined market share of over 95%: VMWare, Hyper-v, Xen and KVM [28]. Unfortunately, the most popular of these -VMWare and Hyper-v- are closed source. This means empirical results can not be verified by analyzing VMM source code. Therefore, any detailed analysis of virtualization overhead is best limited to systems employing Xen or KVM. Because previous studies have shown that KVM is in general by a narrow margin slower than Xen for CPU-bound workloads [7], KVM was picked as the VMM for all experiments presented in this dissertation. This ensures that experimental results are not overly optimistic while at the same time being safely generalizable to other VMMs.

[1] `https://linux.die.net/man/1/taskset`

For the guest OS, Linux is an obvious choice since it is by far the most popular server OS, with the only noteworthy competitor being Windows [121]. The latter is however closed-source, making analysis of results again difficult. Moreover, intuitively the guest OS is not a major contributor to virtualization overhead. As such, results collected under Linux are representative for real-world applications in general.

Because certain forms of overhead only appear when hardware resources are oversaturated at VMM level [110], virtualization overhead should be separately evaluated when the hardware is not shared between $V(S_w)$ and other tasks on the one hand and when it is on the other. These two scenarios may be respectively referred to as undercommitted (UC) and overcommitted (OC). Setting up the UC scenario does not require special considerations. The OC scenario however is a more complicated matter, since without careful system configuration γ is unknown. Moreover, unfair resource allocation is a known issue for synchronization-heavy virtualized workloads [50]. Finally, it is unclear how to attribute some VMM operations to individual VMs. For example, if one VM uses 90% of the system's resources and another uses only 10%, should VMM scheduling overhead be attributed for 90% to the first VM because it uses most of the resources or for 50% because scheduling is only necessary because of the presence of multiple VMs, for which both VMs are equally responsible, irrespective of their resource usage? All of these issues may be avoided by creating an OC environment with exactly two identical VMs, running identical workloads, pinned to the same pCPU set. When both VMs demand all available resources, each will receive 50% thereof. Thus, $\gamma_v = \frac{\gamma_p}{2}$. Cycles used by the VMM may also be attributed equally to each VM, so that $C_v = \frac{C_{VMM}}{2} + C_{VM} = \frac{C_{sys}}{2}$.

Concretely, the host system used for all experiments presented in this dissertation (unless stated otherwise) is a HPE ProLiant DL385 Gen10 server with four Intel Xeon Gold 6138 20-core processors and 256GB of memory. Hyperthreading was disabled, as were C states deeper than C1 to prevent performance degradation due to CPU power management [122]. Ubuntu Server is the OS for both the host and the guest, as it is one of the most popular Linux distributions at the time of writing [123]. CPU count is varied between 4 and 64 and NUMA socket count between 1 and 4. Both UC and OC scenarios are considered. Details may however vary for individual experiments. If so, this is obviously clearly stated where appropriate.

3.3.2 Workloads

Selecting a representative set of workloads is as important as using the correct system settings when evaluating virtualization overhead. Firstly, the set of workloads should be sufficiently broad because even within specific categories of applications individual workloads may vary greatly in nature. Moreover, the workloads are preferably realistic, rather than synthetic programs designed to test a specific system mechanism. Ideally, an existing benchmark suite meeting these requirements should be employed.

Since this dissertation focuses on multithreading, multithreaded, computation-intensive workloads are a natural choice. Several benchmark suites of precisely such workloads exist. Among these, perhaps the most widely used one is Parsec 3.0 [124]. The 13 workloads of this benchmark suite thus serve as the baseline for this dissertation. All of these are compiled using `pthreads` and run with their native inputs. The level of parallelism is set equal to the number of CPUs configured for each test. Wherever appropriate, these workloads are supplemented by other benchmark suites such as SPLASH2X [124] and Phoronix [125].

3.3.3 Measurement

Besides careful preparation, precise collection and processing of data is paramount in order to accurately quantify virtualization overhead. Firstly, immediately before each experiment, the VM should execute a 'warm-up' run of the benchmark to be executed. This pre-warms the OS buffer cache, so that I/O operations are reduced to the absolute minimum. Furthermore, it is almost impossible to guarantee that S_v and S_w remain constant between executions due to non-deterministic aspects of the system (e.g. interrupts, background processes,...). To reduce the variance in S_v and S_w to negligible levels, experimental results should always be averaged over many iterations. Unless stated otherwise, all results shown in this dissertation are derived from 10 iterations of the experiment in question. `Perf`[2] is the standard profiling tool used to collect data.

Since in §3.1 $P(S_w) \cong V(S_w)$, C_p and t_p refer to undercommitted native execution, even when S_v includes overcommitting the system. This is conceptually sound, since multiplexing system resources between $V(S_w)$ and other tasks is opaque to the VM and thus a virtualization effect from the perspective of the workload. On the other hand, this intertwines the effects of virtualization in se and hardware consolidation, which may in general improve

[2] `https://man7.org/linux/man-pages/man1/perf.1.html`

resource efficiency regardless of the technique employed to achieve it. To address this, the regular UC and OC virtualization overhead numbers may be supplemented by an additional 'overcommitted base 2 (OC_2)' value wherever appropriate. This value directly compares C_v and t_v for two concurrent VMs, each running one instance of W to C_p and t_p when executing two concurrent instances of W on $P(S_w)$.

All experimental results are analyzed for each system configuration and each benchmark independently, as is common practice in the field. While this method provides detailed insight into individual results, it does not directly allow for broad conclusions to be drawn with a high degree of certainty regarding the magnitude of the identified virtualization overhead. Doing so would require a detailed statistical analysis, which would in turn require a large amount of expertise and time to conduct properly, which were unfortonately not available within the scope of the Ph. D. project this dissertation documents. Therefore, all results presented in this dissertation are to be viewed as indicative and thus as 'evidence for' rather than 'proof of' the trends these results express.

Because of the limitations of the method employed to gather empirical data, it is of crucial importance that all findings are verified from a technical perspective. Concretely, every trend observed in experimental results must be linked back to some relyably observable system behavior. This can be done by analyzing system source code and hardware features or profiling workload properties. In this way, even though none of the experimental observations can be conclusively accepted, the author is convinced the results presented in this work at least provide credible evidence for the claims they aim to support.

3.3.4 Threats to Validity

Like any empirical work, quantifying virtualization overhead through controlled experiments is liable to threats to validity, which have to be taken into account when interpreting results [126, 127]. These validity threats are often grouped into four categories: internal validity, external validity, construct validity and conclusion validity. Wohlin et. al. provide a detailed breakdown of all possible validity threats based on these four categories [127]. This section details the validity threats applicable to the method outlined above based on their work.

Internal Validity

Internal validity threats pertain to the possibility that the outcome of the experiments does not reflect the effect of the variable(s) the experiment intended to study. When studying virtualization overhead, it is possible that the observed overhead is caused by some other effect rather than virtualization itself. When studying the effectiveness of certain mitigation techniques, it is possible that the mitigation technique in question (or at least the concept upon which it is based) is not responsible for the observed result. Concretely, the following threats must be considered:

- **History/maturation**: Modern systems adapt themselves to the nature of the workload they execute in a variety of ways. Examples include the buffer cache which makes sure disk reads are much faster after the first iteration of a given workload, certain memory allocator heuristics that edapt block sizes to workload demand, etc. This may influence results significantly if not properly controlled for. This work controls for the effects of the buffer cache by disregarding the first iteration of each workload execution and for other adaptive system behavior by performing the same number of iterations for all experiments;
- **Instrumentation**: While the tools used to collect system data (`time`, `perf`, etc.) are generally highly accurate, `perf` in particular is sensitive to overloading under heavy system use. Luckily however, `perf` logs any occurance of overloading so that data tainted by this issue can be discarded and the experiments yielding said data repeated;
- **Ambiguity of direction of causal influence**: This issue is critically important regarding virtualization overhead. Namely, it is well-known that certain negative system effects may cause other positive system effects. A prominent example is some form of virtualization overhead slowing down application progress, which in turn reduces lock contention and thereby reduces issues such as lock holder preemption. This makes it extremely challenging to quantify the impact of any particular form of virtualization overhead on the system. Luckily, as argumented above, this work does not seek such exact quantification. Even though various forms of overhead may interact with each other, it is highly unlikely that these interactions are so severe that they would completely hide certain forms of overhead or completely mitigate the performance benefits of a certain mitigation technique.

External Validity

External validity threats are concerned with the generalizability of experimental results. After all, even a perfectly designed experiment has very limited use if its results can not be used to predict phenomena in the real world. For this work specifically, external validity is a major challenge because of the threats outlined below:

- **Interaction of selection and treatment**: As argued in §3.3.2, selecting a set of workloads representative of the entire set of multithreaded applications currently in use is hardly possible. Therefore, this work opts to employ widely used benchmark suites designed to cover a broad spectrum of application domains for multithreading. This however means that experiments performed with these benchmark suites can not be used to draw quantitative conclusions regarding the population. However, these benchmark suites can be used to indicate that certain causes of virtualization overhead exist or that certain mitigation techniques do have the potential to benefit at least some multithreaded workloads. Thus, the experiments presented in this work are indicative and explanatory rather than quantitative.
- **Interaction of setting and treatment**: Even though the high-level causes of virtualization overhead and potential of certain mitigation techniques are conceptual in nature and thus largely independent of system or workload specifics, all experimental results are only valid for the specific system configuration used to perform that experiment. Therefore, while the nature and general behavior of the identified virtualization overhead or performance benefits of mitigation techniques remain constant across system configurations, their exact quantities may vary greatly. Particularly problematic in this regard is the fact that software evolves rapidly and this work was performed over the course of five years. Therefore, some earlier findings may have already been invalidated by the time of publication. These findings are nevertheless still relevant in a pragmatic sense, since improvements at research level typically take several years to trickle down into industry. Regardless, while it is hardly possible to redo all experiments presented in this dissertation every time a new Linux kernel is released (which is almost daily), all older findings presented in this dissertation are verified through analysis of newer kernel versions and/or through performing a sample of the original experiments using a newer kernel.

Construct Validity

Construct validity describes to what extent the design of an experiment conceptually reflects the phenomenon it is attempting to assess. For example, even if an experiment demonstrates a correlation between some system setting and virtualization overhead, it is not necessarily the case that this system setting is the underlying cause of the overhead (i.e. it may exacerbate some internal phenomenon which does cause the overhead and is only partly dependent on that particular system setting). The most important construct validity threats for this dissertation are listed below:

- **Inadequate preoperational explication of constructs**: This threat is mostly applicable to the first objective defined in §1.3, being identifying the leading remaining causes of virtualization overhead for multithreaded applications. Namely, the entire point of this objective is that the construct under consideration—virtualization overhead—is not properly understood at the moment. It is therefore impossible to guarantee beforehand that the chosen methodology is the most adequate available. The possible impact of this threat has however been minimized by rigorously defining virtualization overhead beforehand—albeit without knowing its exact constituents—and studying existing literature on the topic extensively to take all known causes of virtualization overhead into account and encorporate established best practices;
- **Mono-operation bias**: This threat is reflected perfectly by the OC data set. Namely, the construct of virtualization overhead is in the opinion of the author conceptually best represented using this data set, it does conflate the constructs of virtualization and resource consolidation. This effect was mitigated by including the OC_2 data set, allowing for a multi-facetted interpretation of results;
- **Interaction of testing and treatment**: Because much of the experimental data is gathered by performance profiling during workload execution, it is not unlikely that the very act of performance profiling influences workload performance. This issue was however minimized in several ways: some metrics (e.g. execution time) were measured using multiple tools, multiple independent benchmark runs were performed to test independent metrics rather than measuring all metrics at once to minimize load on the system and identical measurements where applied to all benchmarks in all settings;

- **Restricted generalizability across constructs**: Due to the many layers of abstraction in virtualized systems and quasi endless variety of workloads these systems may be tasked with, some causes of virtualization overhead that emerge under very specific circumstances may be overlooked by the experiments provided in this work. Similarly, proposed mitigation techniques may negatively impact some specific workload ore system configuration not represented in the experiments performed here. The impact of these issues is however almost certainly negligible within the broader context of virtualizing multithreaded applications, since the workloads and system configurations employed in this work have been specifically designed to cover a vast range of real-world use cases. Moreover, all experimental results are verified through source code analysis and literature review so that all findings can be linked back to the theoretical construct causing them. If any ambiguity should emerge, specific additional experiments can be set up to provide clarity.

Conclusion Validity

Conclusion validity is concerned with drawing the correct conclusions from the results. Threats in this regard are mainly comprised of the following:

- **Low statistical power**: Because the method described above does not involve statistical analysis, this work is not able to make quantitative predictions about virtualization overhead. However, it can make qualitative statements regarding the nature of the overhead and unveil general trends;
- **Fishing and the error rate**: This threat is particularly important when assessing the effectiveness of novel mitigation techniques. Namely, when designing these techniques, naturally a particular set of variables is taken into account in order to address a specific issue. Naturally, evaluation of the mitigation technique focusses on these variables of interest. However, it is always possible that a mitigation technique unintentionally negatively impacts some other variable which is not tested, leading to the issue going unnoticed. This issue can however be addressed through defining the metrics of interest in such a general sense that any concerning negative impact on the system would in the end be visible in this metric. This is precisely the reasoning of defining system effects in terms of CPU cycles and application effects in terms of execution time in §3.1. In other words, by consistently including CPU cycles and application execution time in the evaluation, any unintended side effects of interest not directly evaluated by the experiment should become apparent;

- **Reliability of measures**: This validity threat was the main driver for developing the definition of virtualization overhead presented in §3.1 and the thorough formulation of the experimental approach this dissertation adheres to above. Namely, without doing so there was no guarantee that whatever measures used in performing experiments would yield meaningful results;
- **Random heterogeneity of subjects**: As described previously, any work measuring benchmark performance is faced with non-determinism inherent to some system components. Particularly system configurations with more than one NUMA node are sensitive to such non-deterministic performance fluctuations, since slight variations in scheduling may heavily influence memory locality and therefore performance. Nevertheless, averaging all results over a sufficiently large number of experiment iterations as suggested above largely nullifies this threat.

3.4 Related Work

The main contributions of this chapter are providing a generally applicable definition of virtualization overhead and a systematic method to measure said overhead through controlled experiments. Regarding both of these issues, an extensive body of existing work is available to draw on. This section elaborates on each and clarifies the distinctions between the work presented in this chapter and these existing efforts.

3.4.1 Defining Virtualization Overhead

While virtualization overhead is a popular research topic, existing studies do not adequately reflect on the concept of virtualization overhead. All of them employ their own ad-hoc interpretation of virtualization overhead, which they usually measure in terms of some general system metrics. The exact metrics used may vary wildly between studies. Table 3.1 lists the studies most closely related to the goal of this dissertation and enumerates the metrics these studies employ to measure virtualization overhead.

Table 3.1 indicates that while indeed existing work varies wildly in terms of the metrics used to measure virtualization overhead, almost all studies include some form of the metrics 'wall time' (short for wall clock execution time) and 'throughput'. These metrics may be mapped directly to the measures this work arrived at to quantify virtualization overhead. Namely, execution time is intuitively related to $\delta\eta_t$. When throughput is interpreted as the amount of useful work the system may perform in a given amount of time, it can be mapped to $\delta\eta_r$. One may even argue that this holds true for all of the metrics in table 3.1: CPU time is almost identical to CPU cycles, operations per second (OPS) is a measure of throughput, cache misses manifest themselves in increased $\delta\eta_r$ and likely increased $\delta\eta_t$ and latency can be interpreted as application execution time, when the application being considered consists of a single unit of work for which the latency is to be measured. This confirms the validity of the model of virtualization overhead presented in this chapter. Nevertheless, it is sometimes useful to include specialized metrics such as latency and cache misses when studying specific workloads or system aspects for which these metrics are widely used in order to facilitate interpretation of results by readers not familiar with virtualization overhead in se.

Besides the validity of the model for virtualization overhead presented in this chapter, the studies listed in table 3.1 also confirm the importance of rigorously defining what virtualization overhead means. Namely, it is often not exactly clear what the studies listed in this table measure exactly. For example, the

Table 3.1: Related studies and the measures they employ to quantify virtualization overhead.

Study	Wall time	CPU time	Throughput	OPS	Cache misses	Latency
[128]	X	X	X	X		
[129]	X	X				
[19]		X	X	X		X
[130]	X					
[131]			X	X		X
[132]	X		X			X
[133]	X					
[10]	X		X			
[134]			X	X		
[135]	X		X			X
[136]	X		X		X	
[137]	X	X	X		X	X
[9]		X	X			X
[20]		X	X			X
[21]	X		X			X
[8]	X	X				X
[7]						X
[44]	X		X			
[46]	X	X		X		
[51]			X		X	
[53]	X				X	
[138]	X				X	
[139]	X					
[140]		X	X			X
[141]				X	X	
[142]		X	X		X	
[143]	X		X			X
[144]	X					X
[145]	X		X	X		X

term 'throughput' by itself can mean many different things and can be measured in many different ways. None of the papers listed in table 3.1 reflect thoroughly on this, which complicates interpreting their results. Moreover, none of these studies discuss the relationship between the metrics they study and virtualization overhead as a whole. This makes it unclear to readers if the presented findings capture the full effect of virtualization overhead. The systematic definition of virtualization overhead this chapter has provided addresses both of these issues.

3.4.2 Empirical Research

While the studies presented in table 3.1 all perform controlled experiments and describe the methodology they employ to perform those experiments, none of them do so in a generalizable manner. On the other hand, plenty of work describes how to conduct empirical studies in a software engineering context [126, 127, 146, 147, 148, 149]. Naturally however, this second category of related work is too broad in scope to be immediately applicable to this dissertation. Therefore, the methodology presented in this chapter represents a necessary merger of the contributions of both of these categories of existing literature by systematically describing a method to empirically assess virtualization overhead, which in itself is a novel contribution.

3.5 Conclusion

Virtualization overhead is the cumulation of all negative performance effects the virtualization process has on the system on the one hand and applications on the other. These effects may respectively be measured as the reduction in resource efficiency ($\delta\eta_r$) and temporal efficiency ($\delta\eta_t$) in the context of multithreaded applications. A wide variety of issues lay at the root of observed virtualization overhead. Many of these issues are well known and for some of them effective solutions have been widely adopted.

Quantifying virtualization overhead is only feasible through controlled experiments. These experiments must be carefully designed in order for the results to be representative. Most critical are the system configuration and workload choice. Results should be obtained for varying CPU counts, NUMA layouts and hardware contention conditions. Because system state is prone to variance, experiments should be iterated at least tenfold.

3.5.1 Personal Contribution

The definition of virtualization overhead presented in this chapter is the result of many discussions between the author of this dissertation and his supervisors. Additionally, the anonymous reviewers of the publication upon which this chapter is based provided valuable feedback which guided the final formulation of the definition as well as the quantification method described above. Moreover, the latter was continually refined by the main author for the duration of the PhD project documented in this dissertation.

Chapter 4

Virtualization Overhead for Multithreaded Applications

This chapter was previously published as part of:
S. Schildermans et al. "Virtualization Overhead of Multithreading in X86 State-of-the-Art & Remaining Challenges". In: *IEEE Transactions on Parallel and Distributed Systems* 32.10 (2021), pp. 2557–2570

As stated in chapter 1, studying virtualization overhead induced by multithreading is critical in the modern era of cloud-driven HPC. This chapter contributes to this effort by addressing the first pair of secondary research questions outlined in §1.3. Specifically, it provides an overview of the state of the art regarding hardware-assisted virtualization of multithreaded applications in x86 and identifies major outstanding issues in this regard.

Perhaps somewhat ironically, this analysis of virtualization overhead for multithreaded applications starts with a brief study of sequential workloads in a virtualized setting in order to clearly frame results for their multithreaded counterparts presented later within a broader context. All experiments are based on the prescriptions provided in chapter 3. In the interest of generality, the set of analyzed workloads consists of both the PARSEC and SPLASH2X benchmark suites throughout this chapter. All findings are verified through source code analysis and literature review. Moreover, a deeper understanding of the identified virtualization overhead is provided by linking it to its underlying causes.

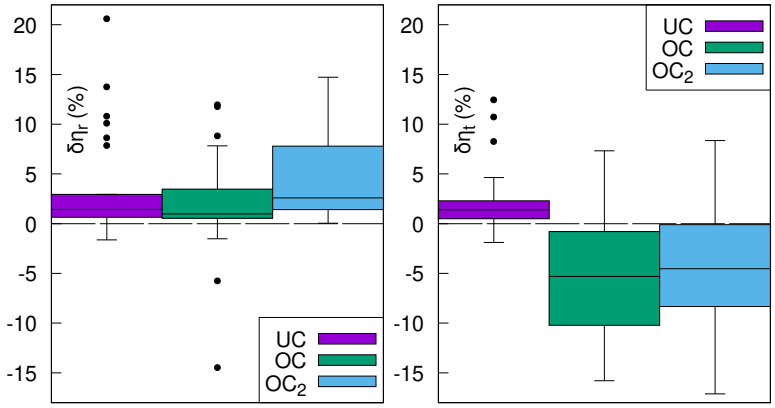
While it can not be guaranteed that all findings presented in this chapter are universally applicable, a wide variety of system configurations and workloads are covered to minimize the threats to validity inherent to empirical work such as this. Moreover, a reflection on how the results shown below would translate to other system configurations is included wherever appropriate.

Main Findings & Contributions

- With the latest virtualization support, overhead imposed on individual threads is low. For sequential applications, overhead is mainly incurred by handling I/O;
- Virtualization overhead for multithreaded applications has been significantly reduced in recent years thanks to various advancements in virtualization technology;
- Multithreaded computations still suffer significant virtualization overhead, especially when the system is overcommitted. Thus, further improvements are desirable;
- For multithreaded applications, there can be a large divergence between system and application effects induced by virtualization. The major driver of this divergence is whether or not the overhead is incurred on the critical path of the application;
- Most virtualization overhead incurred by multithreaded applications is caused by interaction between threads, in the form of data sharing (especially in NUMA systems) and synchronization (especially spinning at user level and blocking synchronization);
- Most multithreaded workloads benefit from being consolidated using virtualization. Some even consume less resources when consolidated.

4.1 Sequential Applications

This section briefly analyzes virtualization overhead for sequential workloads in order to ease interpretation of results for their multithreaded counterparts. To this end, experiments were performed in accordance with the prescriptions given in §3.3. The chosen workloads are the PARSEC and SPLASH2X benchmark suites with the level of parallelism set to one, executed in a VM with a single vCPU. Figure 4.1 shows the results as an aggregate of all the tested benchmarks.



(a) $\delta\eta_r$ (b) $\delta\eta_t$

Figure 4.1: Box plots of virtualization overhead for the sequential versions of all PARSEC and SPLASH2X workloads, aggregated for each scenario.

As figure 4.1 shows, modern improvements to virtualization technology have minimized virtualization overhead for sequential workloads. On average, both $\delta\eta_r$ and $\delta\eta_t$ are negligible for the tested benchmarks. Some outliers can be observed however. Detailed analysis reveals that these are attributable to I/O. This is a well-known issue, as described in §3.2.3.

Generally, $\delta\eta_r$ is greater than $\delta\eta_t$ in figure 4.1. In the OC scenario, $\delta\eta_t$ is even negative. Upon closer analysis, QEMU was found to be responsible for this, as it has to handle write-backs of newly generated data (reads come from the pre-warmed OS buffer cache). This consumes up to 20% of the CPU resources used by the entire workload. Because QEMU runs on a separate host thread in parallel with the VMs, this does not increase $\delta\eta_t$. On the contrary, this effect results in a negative $\delta\eta_t$ in the OC scenario since the second VM may run while the first is waiting for QEMU.

4.2 Multithreaded Applications

Evaluating virtualization overhead for multithreaded applications requires more consideration than doing so for sequential workloads, as stated in §3.3.1. Specifically, a variety of system configurations is to be considered. For the analysis presented in this section, the following vCPU and NUMA settings were evaluated:

- 4 CPUs, 1 NUMA node,
- 8 CPUs, 1 NUMA node,
- 16 CPUs, 1 NUMA node,
- 32 CPUs, 2 NUMA nodes,
- 64 CPUs, 4 NUMA nodes.

In each of the above scenarios, all PARSEC and SPLASH2X workloads were evaluated with the level of parallelism set equal to the number of CPUs available in the respective configuration. Figure 4.2 shows the results in a manner analogous to figure 4.1 for each studied system configuration separately.

From figure 4.2 it is clear that much like for sequential workloads, $\delta\eta_t$ is limited in general for multithreaded applications. In the OC scenario it is even strongly negative; increasingly so as vCPU count increases. Firstly, this is caused by processing I/O in the background, as described in §4.1. Secondly, the pair of vCPUs competing for each pCPU can compensate for each other's idle time. Namely, when a vCPU is idle in the UC scenario, the pCPU hosting that vCPU is also idle. In the OC scenario however, a vCPU from another VM can perform useful work during this time, which naturally increases system throughput. This is confirmed by the OC_2 data set, since for this data set $\delta\eta_t$ is positive, as in a native setting this consolidating effect also occurs.

Figure 4.2 also shows that multithreaded applications still suffer high virtualization overhead compared to sequential ones. This overhead tends to greatly increase with vCPU count, indicating that mitigating it will only gain importance as time goes on, since VMs are likely to follow physical systems in sporting ever larger numbers of CPUs [150]. However, figure 4.2 simultaneously indicates that great advancements have been made in the past few years with regard to mitigating virtualization overhead for multithreaded applications. For example, a study from 2015 found that the performance of the *Dedup* benchmark could be degraded by more than 500% in an overcommitted virtualized environment relative to native execution [10]. Given that no benchmark in figure 4.2 suffers a $\delta\eta_r$ of more than 175% and a $\delta\eta_t$ of more than 80%, these at first glance concerning performance numbers are at the same time pleasing within the broader context of the field.

When comparing figures 4.1 and 4.2, the variance in virtualization overhead between benchmarks appears to be much greater for multithreaded applications than for sequential ones. For some benchmarks $\delta\eta_r$ is strangely negative, while for others it may be over 150%. To better understand this, figure 4.3 provides a detailed breakdown of the average and maximum $\delta\eta_r$ by individual benchmark,

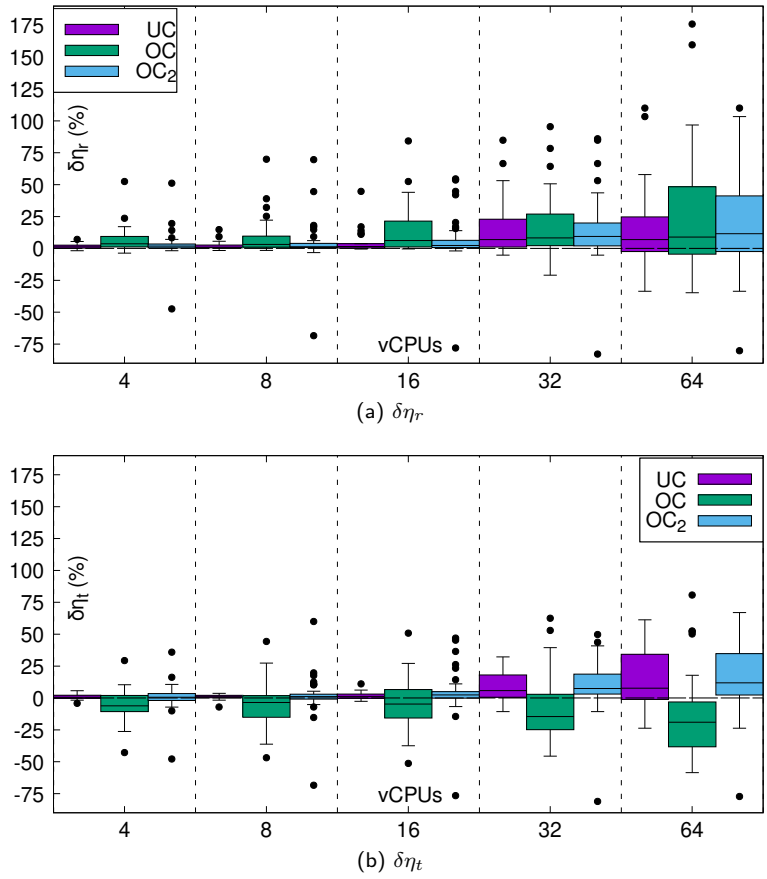


Figure 4.2: Box plots of virtualization overhead for various multithreaded executions of PARSEC and SPLASH2X workloads. Results for all benchmarks are aggregated for each scenario.

aggregated for all vCPU counts with overlapping bars. These bars are split into cycles spent at guest and host level, respectively. Similarly, figure 4.4 shows the average and maximum values of $\delta\eta_t$ for each benchmark with overlapping bars, aggregated for all studied vCPU configurations.

Figures 4.3 and 4.4 provide several insights. Firstly, the OC_2 data set explains why $\delta\eta_r$ is negative for some benchmarks in the OC scenario (e.g. *FFT*, *Radiosity*, *s.Raytrace*). Namely, overcommitting has a positive effect on η_r in a native setting as well. This is thus an effect of consolidation rather than virtualization. The main causes of this effect are the following:

- **Reduced lock contention**: As the system is overcommitted, the effective CPU utilization of individual benchmarks is lower. As less threads are competing for the same synchronization constructs, less cycles are wasted;
- **NUMA management**: When the system is overcommitted, the scheduler can do a better job of balancing the workload between different NUMA nodes, thus reducing memory latency;
- **Reduced idling**: When a CPU runs out of work, the OS performs several operations to prepare it to enter an idle state. When the system has more work, it is less likely to start idling, thus eliminating these operations.

Secondly, the relationship between $\delta\eta_r$ and $\delta\eta_t$ is not simply linear for multithreaded workloads, even in the UC scenario. To better understand this at first glance unintuitive finding, we define the 'overhead impact factor (ω)' as a measure of the correlation between system effects and application effects:

$$\omega = \frac{1 + \delta\eta_t}{1 + \delta\eta_r}$$

Intuitively, ω shows to what extent system-level virtualization overhead has an observable impact on application performance. Studying this metric yields several interesting findings. Firstly, ω is for almost all studied workloads smaller than 1. This indicates that $\delta\eta_r > \delta\eta_t$ or in layman's terms that not all system-level overhead is observable by end users. This general trend may in part be explained by the following observations:

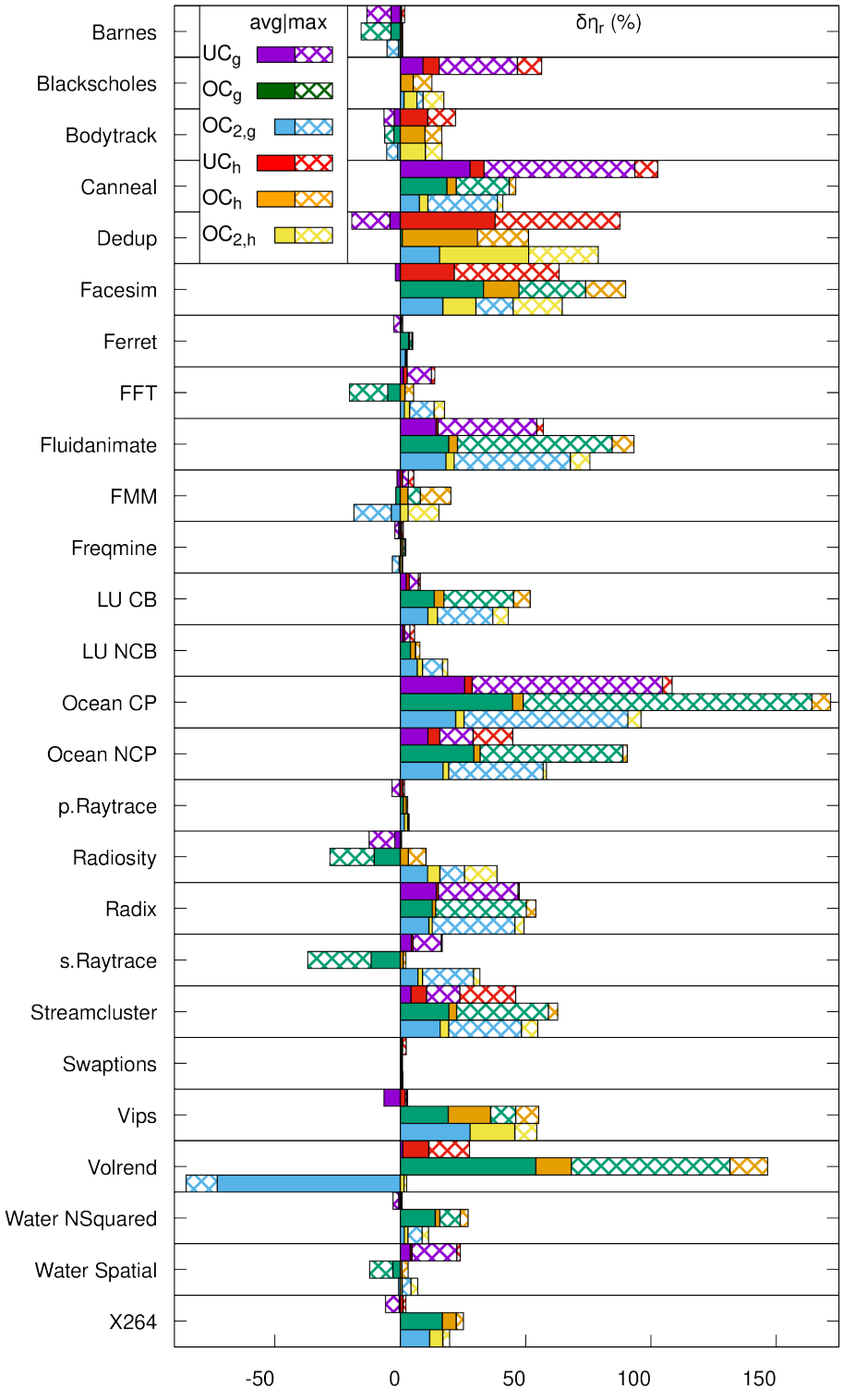


Figure 4.3: Average and maximum $\delta\eta_r$ for the studied vCPU counts, displayed separately for each benchmark with overlapping bars.

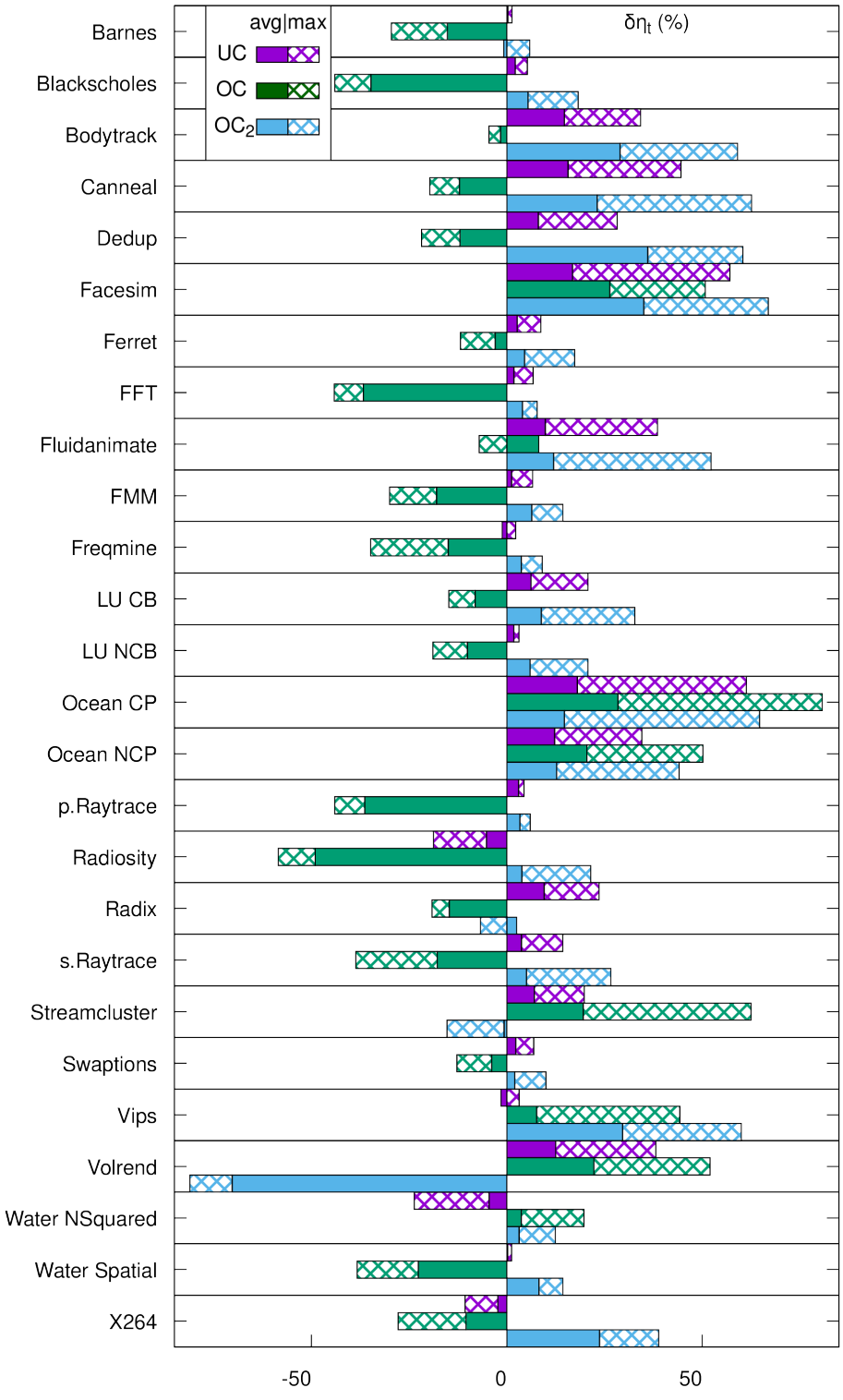


Figure 4.4: Average and maximum $\delta\eta_t$ for the studied vCPU counts, displayed separately for each benchmark with overlapping bars.

- The virtualized benchmarks show higher CPU utilization than their native counterparts, caused by e.g. I/O offloaded to QEMU. Previous research has shown that hardware-assisted I/O virtualization techniques such as SR-IOV (see §2.2.4) -while improving performance- actually increase CPU usage [140].
- Since many of the system effects introduced by virtualization involve CPU-intensive operations (e.g. instruction emulation), they push said CPU to its highest boost frequency (185% of the nominal frequency in the case of the test system employed for this work). The average CPU frequency is therefore higher in a virtualized context.

Moreover, for multithreaded applications, the variance in ω ($\sigma\omega$) between benchmarks is very high. For example, for *Bodytrack*, UC $\omega \approx 1.1$, while for *Ocean CP*, OC $\omega \approx 0.6$. This can be explained by the fact that the execution time of a multithreaded application is determined solely by its critical path [151]. In brief, the critical path is the execution path taking the largest amount of time to complete. For example, consider an image processing application employing 10 threads, the first 9 of which process an equally sized section of the image, while the last thread processes a section twice that size. Assuming processing time is directly proportional to image section size, the critical path of the application is intuitively the tenth thread. Even when the workload of the other 9 threads is doubled, the execution time of the application remains identical (assuming ample system resources are available), despite the resources consumed by the application increasing by 82%. Conversely, if the workload of the tenth thread is doubled, application execution time doubles, despite the resources consumed by the application only increasing by 18%. For virtualization overhead this means that when $\delta\eta_r$ is located mostly on the critical path, $\delta\eta_t$ increases drastically. Otherwise, $\delta\eta_r$ may have little to no effect on $\delta\eta_t$. To illustrate this, figure 4.5 shows the distribution of cycles over individual CPUs for the *Bodytrack* and *Ocean CP* benchmarks in both a native and virtualized setting, with 64 CPUs spread over 4 NUMA nodes in an UC scenario. The results are normalized to the native execution so that $\sum_{C=0}^{63} P(C) = (\delta\eta_r + 1)(\times 100\%)$, with C a particular CPU ID.

Figure 4.5 shows that system-level overhead is distributed very differently between vCPUs for *Bodytrack* and *Ocean CP*. Regarding the former, none of the vCPUs show much overhead, except for one. It is likely other vCPUs will at some point have to wait for this overhead-heavy vCPU because it is under such a heavy load, thus slowing down the entire application. Regarding the latter on the other hand, the distribution of $\delta\eta_r$ is much more egalitarian. Because of this, many of the system effects are likely not part of the critical path, yielding a much smaller ω.

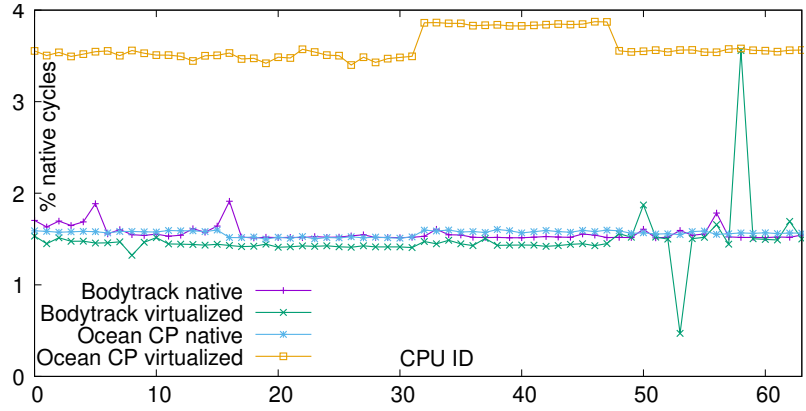


Figure 4.5: Distribution of cycles over CPUs for the 64 CPU variants of *Bodytrack* and *Ocean CP*, normalized to native execution.

While figure 4.5 explains how the nature of $\delta\eta_r$ may affect $\delta\eta_t$ differently depending on the workload, a more in-depth analysis is needed to explain what causes this difference in nature to begin with. Namely, knowing that $\delta\eta_r$ may have many different causes (see §3.2), it is clear that for each workload, $\delta\eta_r$ is constituted of a unique combination of distinct factors that each influence $\delta\eta_t$ (and thus ω) in a different way. Figure 4.3 provides some indication of this variance in composition of $\delta\eta_r$, as the ratio of host- and guest-level overhead varies between applications. This is thus a good starting point to gain a deeper insight into the constituents of $\delta\eta_r$ for multithreaded applications. Specifically, based on figure 4.3, the benchmarks may be grouped in four different categories depending on the nature of $\delta\eta_r$:

- **Negligible overhead**: *Barnes*, *Ferret*, *FFT*, *FMM*, *Freqmine*, *LU NCB*, *parsec.Raytrace*, *Radiosity*, *splash2x.Raytrace*, *Swaptions*, *Water NSquared* and *Water Spatial*;
- **High guest overhead**: *Blackscholes*, *Canneal*, *Fluidanimate*, *Ocean CP*, *Ocean NCP* and *Radix*;
- **High host overhead**: *Bodytrack*, *Dedup*, *Facesim*, *Vips* and *Volrend*;
- **High overcommitted overhead**: *LU CB*, *Streamcluster*, *Vips*, *Volrend*, *X264*.

Note that some benchmarks exhibit characteristics of several overhead profiles and were therefore added to multiple categories. Below each of these categories is discussed in detail. Because figure 4.2 indicates that overhead varies severely between VM sizes, the discussion of each category begins with a breakdown of the overhead for each VM size in the most relevant scenario. This allows for reasoning about the most likely causes of the overhead for that category. This reasoning is subsequently reinforced with further suitable empirical evidence as needed.

4.2.1 Negligible Overhead

About half of the tested benchmarks do not exhibit significant virtualization overhead. This shows that even for workload groups which are by their nature considered to be prone to virtualization overhead such as the studied multithreaded applications, modern virtualization techniques are often highly efficient. Moreover, this data shows that virtualization overhead is highly dependent on the specific workload and even groups of applications sharing many high-level characteristics may exhibit wildly varying performance.

4.2.2 High Guest Overhead

The benchmarks displaying high guest overhead show strongly varying behavior depending on system settings. Firstly, several of these benchmarks display most overhead in the UC scenario, while others show higher OC overhead in figures 4.3 and 4.4. However, the OC_2 data set is for the latter group similar to the UC one, indicating that even on physical systems, overcommitting adds overhead for these benchmarks. The increase in OC overhead is thus due to resource consolidation rather than virtualization. Therefore, analyzing virtualization overhead in the UC scenario is sufficient for this category of benchmarks. In light of this, figure 4.6 shows a breakdown of the benchmarks showing high guest overhead for each analyzed system configuration in the UC scenario.

In figure 4.6, overhead is negligible for all system configurations employing only one NUMA node. For configurations with multiple NUMA nodes on the other hand, overhead increases dramatically. This makes NUMA an obvious suspect regarding the underlying cause of the virtualization overhead these benchmarks incur. Namely, memory-intensive applications may often access data on remote NUMA nodes. As outlined in §3.2.8, in a VM the scheduler is unaware of the NUMA configuration of the physical hardware, preventing it from optimizing NUMA locality like it would natively. For computation-intensive workloads such as the ones employed in this study, analyzing cycles per instruction (CPI)

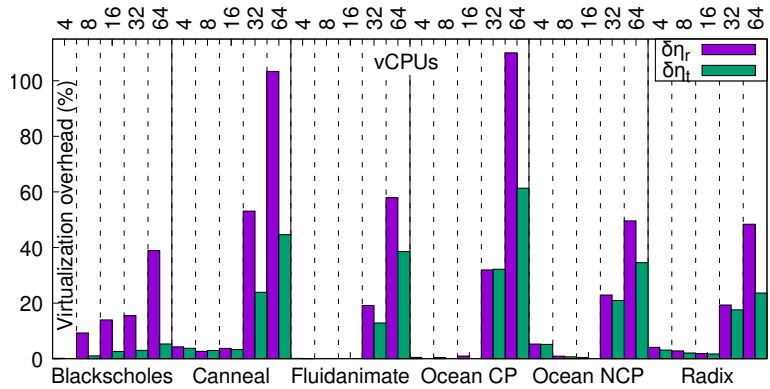


Figure 4.6: Breakdown of virtualization overhead for the benchmarks with high guest overhead in the UC scenario.

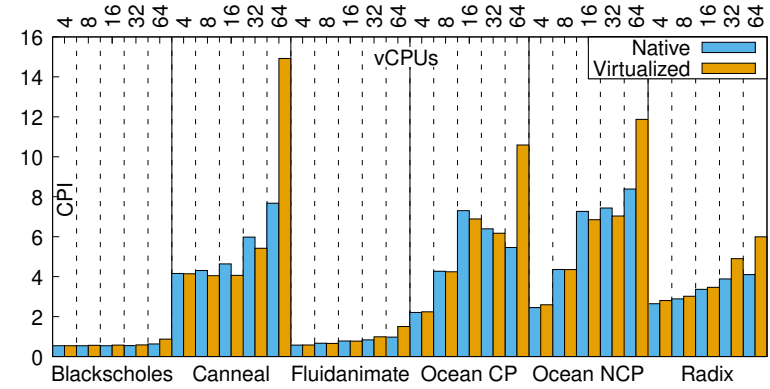


Figure 4.7: CPI for the benchmarks displaying high guest-level overhead in the UC scenario, broken down per vCPU count.

can prove this hypothesis, since it indicates memory latency [152]. As such, figure 4.7 shows the CPI for each combination of workload and system settings in figure 4.6 in both native and virtualized contexts.

Figure 4.7 verifies the above conjecture. Overhead is highest for the benchmarks with the highest CPI, being the most memory-intensive benchmarks. For native executions, CPI increases slightly with CPU count. When virtualized, this increase is much more pronounced, particularly in the scenario with 64 vCPUs spread over 4 NUMA nodes. *Ocean CP* is the only exception. However, detailed analysis shows that this benchmark is bottlenecked by memory bandwidth.

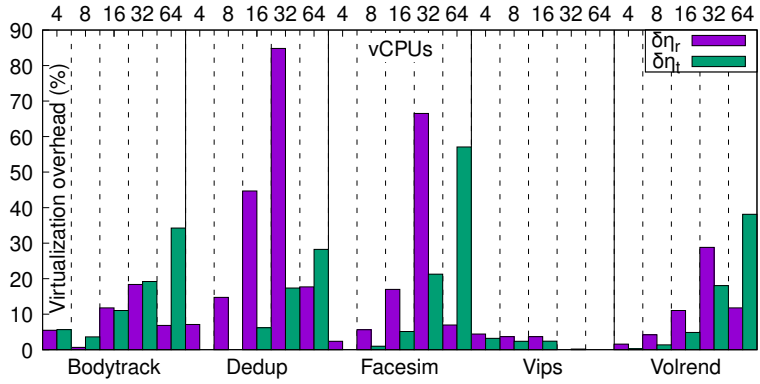


Figure 4.8: Breakdown of virtualization overhead for the benchmarks with high host overhead per CPU count in the UC scenario.

When more NUMA nodes are used, available bandwidth increases, improving performance despite increased memory latency.

For all benchmarks in figure 4.6, ω is low. The reason for this is that performance-critical data tends to be accessed often and thus cached. Only data that is rarely used is fetched from main memory, which is usually input for worker threads and therefore not likely to be directly on the critical path.

Abstraction of the underlying system is a core concept of virtualization, implying that the above issue is independent of the virtualization technology used. Rather, it depends on the host system $P(S_v)$. All popular virtualization platforms are consequently known to struggle with NUMA locality [153, 154].

4.2.3 High Host Overhead

Figures 4.3 and 4.4 indicate that most of the benchmarks suffering high host-level virtualization overhead are mostly affected in the UC scenario. Those that do not (*Vips* and *Volrend*) are also included in the 'high overcommitted overhead' category, which is elaborated on below. Therefore, this section focusses on the UC scenario, only discussing OC results when specifically required to provide a complete insight in host-level virtualization overhead. As such, figure 4.8 shows a breakdown of the virtualization overhead for the benchmarks suffering high host overhead for all studied system configurations in the UC scenario.

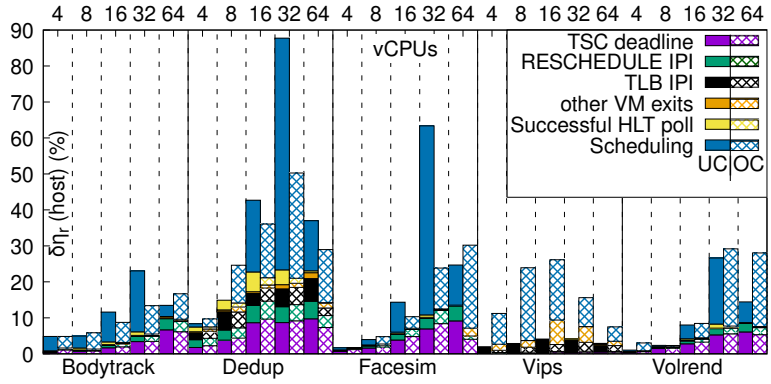


Figure 4.9: Breakdown of host cycles for the benchmarks with high host overhead into their main causes per vCPU count.

The results displayed in figure 4.8 are interesting. $\delta\eta_r$ rises steadily with CPU count until 32 CPUs, after which it drops drastically. $\delta\eta_t$ however continues to rise for all benchmarks with the exception of *Vips*. ω thus varies greatly between benchmarks and CPU counts. It is therefore obvious that a further breakdown of these results is necessary. Since any host operations are preceded by a VM exit for systems based on hardware-assisted virtualization, it makes sense to perform this breakdown based on CPU cycles spent on different kinds of VM exits. Figure 4.9 shows exactly this. Note that in contrast to figure 4.8 both the UC and OC scenarios are included in this figure, since this may provide additional insight in the nature of the host-level virtualization overhead, even though the main interest of this section lies with the UC scenario.

Figure 4.9 explains the variance in ω observed in figure 4.8. Namely, the strange pattern for $\delta\eta_r$ is exclusively attributable to scheduling. When cycles spent on scheduling are ignored, one observes a consistent, high ω. This is logical, since in the UC scenario, VMM-level scheduling almost exclusively occurs when the VM voluntarily yields a vCPU. Therefore, host-level scheduling is rarely part of the critical path. Most other VM exits on the other hand are attributable to the guest attempting to perform some sensitive operation requiring VMM involvement. Many of these are by nature highly likely to be on the critical path, thus yielding a high ω. Below, all of these VM exits are discussed in detail in terms of their high-level causes.

Blocking Synchronization

Blocking synchronization is prevalent in multithreaded applications, as discussed in §3.2.6. The same section notes that while highly efficient in a native context, this synchronization mechanism is known to induce significant host-level virtualization overhead through vCPU scheduling, the BWW problem and IPIs. Additionally, figure 4.9 reveals another complication arising from virtualizing blocking synchronization, which has to the best knowledge of the author never been described in literature. Namely, all popular operating systems update the global system time through a mechanism called the 'scheduler tick', which consists of periodic per-CPU timer interrupts, in the case of Linux preferably driven by the CPU's time stamp counter (TSC). Because this scheduler tick is relatively resource-intensive, modern kernels tend to disable it when the CPU is idle. Specifically, when a CPU is about to enter an idle state, the kernel attempts to heuristically predict how long this idle state will last. If it is determined to likely be sufficiently long, the tick is deferred until the next scheduled timer or read-copy-update (RCU) event or, if none are available, disabled entirely. When the CPU is awoken again, the original tick frequency is restored [115]. This is called 'tickless kernel mode' and yields energy savings of up to 70% relative to a classic naive periodic tick [155]. However, altering the scheduler tick requires writing to the `TSC_DEADLINE` MSR, which induces a VM exit. This explains the VM exits due to `TSC_DEADLINE` MSR writes shown in figure 4.9.

All of the virtualization overhead induced by blocking synchronization follows a predictable pattern. Namely, when a thread blocks on a contended lock and there are no other runnable tasks for the vCPU, the guest kernel usually disables its scheduler tick and issues a `hlt` instruction, resulting in two VM exits. When the thread is woken up again, two more VM exits likely follow for firstly sending a `RESCHEDULE` IPI to an idle vCPU in order to schedule the newly awoken thread and secondly reactivating the scheduler tick on that vCPU. Thus, each blocking operation results in up to four VM exits. Figure 4.10 shows all of this schematically.

Figure 4.9 shows that each of the operations inducing VM exits displayed in figure 4.10 can be costly. Especially surprising is the fact that `TSC_DEADLINE` MSR writes account for a $\delta\eta_r$ of up to 10%, since tickless kernels have been described before in literature as having a positive effect on virtualization [156]. Nevertheless, figure 4.9 shows that scheduling, which is almost always triggered by a `hlt` VM exit, dwarfs any other cause of host-level virtualization overhead for most studied benchmarks. Much of this cost may be attributed to halt polling (see §3.2.6), which has several interesting implications with regard to virtualization overhead:

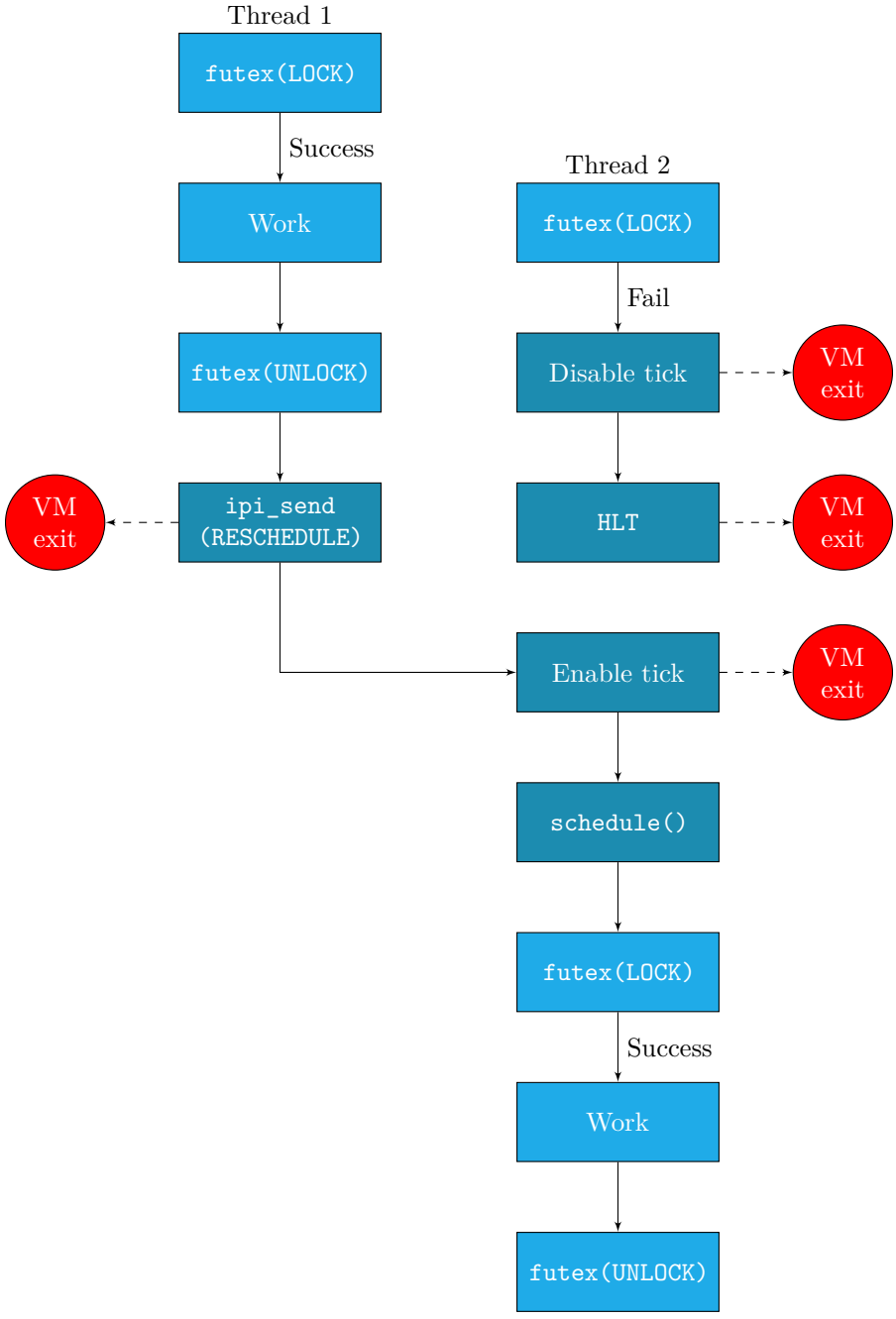


Figure 4.10: Schematic overview of the execution flow of two threads contending for a blocking lock in a state-of-the-art virtualized environment. User space operations are shown in light blue, kernel operations in dark blue.

- When halt polling is successful (i.e. the vCPU is woken up before the polling ends and is immediately rescheduled), the cost of handling `HLT` VM exits is limited. When it is unsuccessful on the other hand (i.e. the polling interval expires and the vCPU needs to be descheduled anyway), the cost of handling `HLT` VM exits becomes very high. Because cycles spent on unsuccessful polling only slow down the scheduling process, they are considered to be scheduling overhead as well in figure 4.9;

- $\delta\eta_r$ is in general much higher for the system configuration with 32 vCPUs than for that with 64 vCPUs in figure 4.9. This is a consequence of the heuristics KVM uses to manage the polling threshold. If the poll was unsuccessful, KVM grows or shrinks the threshold if the vCPU was blocked for resp. a short or long time [115]. As vCPU counts increase, so do contention and average blocking time, which in turn increase the polling threshold. At 64 vCPUs however, the average blocking time is so long that the polling threshold shrinks to 0. We confirmed this by measuring the success rate of halt polling for the studied workloads under different system configurations, which drops from 30% on average for 4 vCPUs to close to 0% for 64 vCPUs;

- Halt polling is largely responsible for the strange evolution of ω in figure 4.8. By design, halt polling expends CPU cycles to improve performance, lowering ω ever more as the polling threshold grows with vCPU count up to 32 vCPUs. When the polling threshold shrinks back to 0 for 64 vCPUs, ω rises drastically as $\delta\eta_r$ drops at the expense of $\delta\eta_t$;

- $\delta\eta_r$ is higher in the UC scenario than in the OC scenario in figure 4.9. This can be explained by the fact that contrary to the UC scenario, halt polling can degrade system throughput in the OC scenario because upon a `HLT` VM exit, the host most likely has other runnable tasks ready to be scheduled on the yielded CPU, which makes spending cycles on polling a pure waste time and resources. Therefore, KVM disables halt polling when the CPU has runnable tasks available when a `HLT` VM exit occurs [115], reducing $\delta\eta_r$ in the OC scenario at the cost of increasing application latency.

Host-level virtualization overhead may vary greatly depending on the system configuration. For example, as the root cause of the VM exits induced by `TSC_DEADLINE` MSR writes lies within the guest OS, this overhead may vary between guests. The VM exits themselves however are handled comparably by Xen and KVM, as are those related to sending IPIs. In terms of hardware, Intel and AMD offer unique APIC virtualization extensions (resp. APICv [42] and AVIC [112]). While implementation details differ, their effect and performance

are similar. Both eliminate the need for VMM intervention to inject IPIs and acknowledge their receipt, but still require a VM exit to write the `ICR` MSR. Finally, halt polling overhead may vary drastically between VMMs. In Xen HVM for example, halt polling is not implemented. $\delta\eta_r$ will thus be lower in the UC scenario for Xen than for KVM, while $\delta\eta_t$ will be higher. In the OC scenario on the other hand, scheduling overhead for Xen will be comparable to that for KVM.

Virtual Memory Management

Figure 4.9 shows that *Dedup* and *Vips* spend a lot of resources on processing VM exits induced by TLB shootdowns (see §3.2.7). Analysis of the system calls invoked by these workloads reveals that most of these TLB shootdowns are caused by resizing the heap. Namely, heap resizing involves acquiring memory from or returning memory to the OS, which is done through system calls such as `madvise` and `mprotect`, which in turn invoke TLB shootdown IPIs. While there are other causes of TLB shootdowns such as page migrations, these are insignificant for the evaluated workloads.

The exact amount of heap resizing operations an application induces is highly dependent on its source code and the underlying system libraries it employs. For example, when an application often allocates and frees small amounts of memory, highly memory-efficient memory allocators may immediately return the freed memory to the OS, only to request new memory soon after. The fact that the studied benchmarks all employ glibc's ptmalloc2 as their memory allocator—which is by nature highly memory-efficient—thereby explains the TLB shootdown-related virtualization overhead some workloads exhibit in figure 4.9.

As the overhead induced by TLB shootdown IPIs is handled comparably across hardware platforms and VMMs, similar performance is to be expected for systems from different vendors with otherwise comparable properties.

Spinning at Kernel Level

Some years ago, spinning at kernel level was a serious issue for overcommitted virtualized systems in the form of LHP and related issues, as described in §3.2.5. Figure 4.9 however indicates that PLE is very effective at dealing with this. Only *Vips* in the OC scenario suffers from many PLE VM exits. While the overhead caused by these exits themselves is low, they invoke the scheduler, inducing significant scheduling overhead. As *Vips* incurs negligible `HLT` and

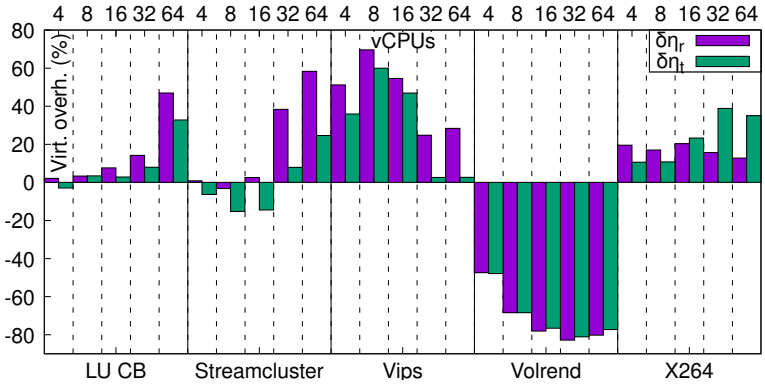


Figure 4.11: Breakdown of the virtualization overhead in the OC2 scenario for the benchmarks that show high overhead in the OC scenario.

preemption timer VM exits compared to the other workloads suffering high host-level virtualization overhead, almost all the scheduling overhead for *Vips* shown in figure 4.9 can be attributed to PLE. Nevertheless, this scheduling overhead can be considered acceptable, since it is comparable to that for other benchmarks in the OC scenario and the scheduler would otherwise be triggered anyway by other mechanisms.

As stated in §3.2.5, AMD's PF is conceptually identical to Intel's PLE [112]. Both solutions are treated equally by KVM. Moreover, Xen source code reveals that it handles both hardware features much like KVM. It is thus fair to conclude that spinning at kernel level has been tackled effectively across hardware and virtualization platforms.

4.2.4 High Overcommitted Overhead

Naturally, the benchmarks only showing significant virtualization overhead when the system is overcommitted are best studied in the OC scenario. As such, figure 4.11 breaks down virtualization overhead for these benchmarks in the OC scenario by CPU count. This figure is based on the OC_2 data set to eliminate the effects of server consolidation.

The results in figure 4.11 are at first glance bewildering. However, upon careful inspection, one may distinguish two subcategories in the presented benchmarks in figure 4.11:

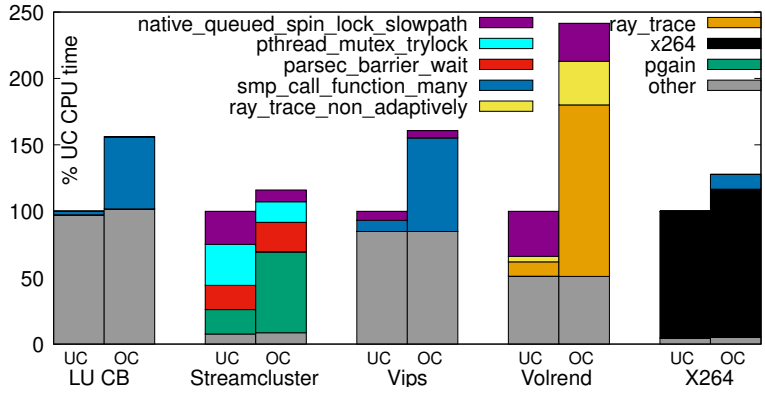


Figure 4.12: Comparison of subroutine CPU profile between UC and OC virtualized execution with 64 vCPUs for the benchmarks displaying high overcommitted virtualization overhead.

- **Positive overhead**: *LU CB*, *Vips*, *X264*,
- **Negative overhead**: *Streamcluster*, *Volrend*. Note that besides overhead related to overcommitting, *Streamcluster* suffers from NUMA locality issues, distorting its results.

In an effort to understand the above patterns, the call stack of the workloads from figure 4.11 was analyzed in detail. Figure 4.12 compares the total CPU cycles spent on each subroutine during virtualized workload execution in respectively the UC and OC scenario. Only the 64 vCPU variants of the workloads were studied, since figure 4.11 indicates that variance between system configurations is limited when accounting for the NUMA-related overhead incurred by *Streamcluster*.

Figure 4.12 shows that for the benchmarks with positive OC_2 overhead in figure 4.11, the system function `smp_call_function_many` is mainly responsible for the difference between UC and OC CPU time consumed by the workload, while for the benchmarks with negative OC_2 overhead some application-level subroutines are the culprit. Both of these groups are discussed in detail below.

Translation Lookaside Buffer Shootdown Preemption

`Smp_call_function_many` is a system-level function used to send IPIs. In the OC scenario, at least some IPIs thus appear to increase in performance cost. Source code analysis reveals that specifically TLB shootdown IPIs are responsible for this. The benchmarks exhibiting positive overhead in figure 4.11 are thus clearly suffering from TLB shootdown preemption (see §3.2.7).

Since TLB shootdown preemption is an example of excessive kernel-level spinning, PLE largely mitigates virtualization overhead associated with this issue. However, as figure 4.11 shows, PLE is not a perfect solution. Namely, the prolonged execution time of the `smp_call_function_many` routine in the OC scenario shown in figure 4.12 is a consequence of the fact that PLE can only trigger a VM exit after some spinning has already occurred. Note that because this spinning takes place in the guest kernel, it is visible as guest-level overhead in figure 4.3.

User-Level Spinning

By analyzing the source code of the subroutines indicated by figure 4.12 as suffering a severe performance penalty in the OC scenario for the benchmarks displaying negative OC_2 overhead in figure 4.11, we found that the common denominator of all these subroutines is that they contain programmer-defined spinning synchronization primitives. Such primitives may lead to a LHP-like problem at user level. Below this issue is illustrated using *Volrend*, since figure 4.11 indicates that this benchmark suffers the most from this issue, which we call 'user-level spinning'.

The `Ray_Trace` subroutine defined in *Volrend*'s source code, which according to figure 4.12 consumes approximately ten times more cycles in the OC scenario compared to the UC scenario, contains the user-level spin-based barrier shown in listing 4.1. Like with classic LHP, in OC scenarios it is possible that a vCPU holding such a custom synchronization primitive is preempted by the VMM, forcing all vCPUs waiting for it to spend exorbitant amounts of time spinning. As shown in figure 4.11, this may lead to catastrophic virtualization overhead. Even more problematic is that PLE can not intervene here, as it relies on the `PAUSE` instruction to work. Programmer-defined synchronization primitives rarely compile down to this instruction. Moreover, PLE only works in kernel mode [42]. As such, user-level spinning is an as of yet unaddressed issue which has to the best knowledge of the author received no attention from scientific literature nor industry.

```
LOCK(Global->CountLock);
Global->Counter--;
UNLOCK(Global->CountLock);
while (Global->Counter);
```

Listing 4.1: User level spin-based barrier in *Volrend*.

Interestingly, while overall performance is clearly degraded in the OC scenario for *Streamcluster* and *Volrend* due to user-level spinning, figure 4.12 shows a decrease in kernel-level spinning (`native_queued_spin_lock_slowpath`) and blocking synchronization (`pthread_mutex_trylock`) for *Streamcluster* due to reduced lock contention in the OC scenario, as fewer effective resources are available to each instance of the benchmark. This illustrates the complexity of quantifying virtualization overhead and categorizing the benchmarks, as a system setting may impact varying overhead constituents in varying or even opposite ways.

It is obvious that application design plays a major role in virtualization overhead due to user level spinning. Notwithstanding, the following system settings may greatly influence the severity of user-level spinning:

- Increasing thread- and vCPU counts leads to more intensive spinning synchronization, as indicated by figure 4.11. This problem will thus gain importance towards the future, as CPU counts tend to grow [150];
- More frequent task switches increase the chance that a thread holding a lock gets preempted, increasing the severity of user-level spinning. Figures 4.3 and 4.4 prove this, as *Volrend* shows high overhead for the OC data set, but negative overhead for the OC_2 data set. Firstly this indicates that user-level spinning is also an issue in a bare metal context. Secondly, the OC virtualized execution is faster than its native counterpart because in each VM only one instance of the benchmark executes, while natively two instances are run within the same OS for the OC_2 data set. As time slices are allocated to vCPUs at a much coarser granularity than to threads, it is much less likely that a lock-holding thread is preempted in a VM, thus reducing user-level spinning.

Previous research has shown that many applications make use of custom user-level spinning synchronization primitives [157]. Given the potential severity of user-level spinning in a virtualized setting and the tendency for vCPU counts to increase towards the future, addressing this issue is paramount. Since user-level spinning originates from the application, it is a conceptual rather than an implementation-related issue from the VMM's perspective. Therefore, all VMMs and hardware are equally prone to this problem.

4.3 Longevity of Results

Like all of the chapters in this dissertation employing empirical methods, the results presented in this chapter are susceptible to the threats to validity listed in §3.3.4. Because the work presented in this chapter was conducted in the earlier stages of the Ph. D. project this dissertation documents, it is prudent to particularly ensure the findings discussed here are still valid. In particular, Ubuntu 18.04.1 was used as both the host and the guest OS, which is based on Linux 4.15, dating back to January 2018. Therefore, a sample of the evaluated benchmarks was re-evaluated using the latest stable Linux release at the time of finalizing the publication upon which this chapter is based (December 2019), namely 4.19.88. The chosen experiment sample consists of one benchmark from each category defined in §4.2, executed with 64 threads/CPUs spread over four NUMA nodes: *Bodytrack* (high host overhead), *Ferret* (negligible overhead), *Ocean CP* (high guest overhead) and *X264* (high overcommitted overhead). All of these yield similar results for the newer kernel, with the exception of *X264* in the OC scenario. In particular, the overhead induced by TLB shootdown preemption has disappeared. After analyzing the Linux kernel patch logs, we found that in kernel 4.16 a patch was implemented that mitigates this problem entirely by paravirtualizing TLB shootdowns in Linux/KVM [158]. Since this patch, the guest only sends TLB shootdown IPIs to vCPUs that are running, while all other vCPUs are flagged to flush their TLB on rescheduling. A similar solution has been implemented more recently for Xen [159].

4.4 Related Work

Studies quantifying virtualization overhead are plentiful. However, most fail to provide deep insight into overhead causes or their link to system and application effects. Indeed, most related work does not even explicitly distinguish between these two forms of overhead, as table 3.1 has made clear in the previous chapter. More profound work on the other hand tends to have a very narrow scope, only addressing a specific issue within the broad landscape of challenges related to virtualization. When narrowing the scope to multithreaded applications, qualitative related work becomes even more scarce. Table 4.1 lists all of said qualitative existing work known to the author which addresses at least some cause of virtualization overhead for multithreaded applications in detail, ordered by publication year. By 'in detail' is meant describing the causes of the overhead in technical depth, as opposed to merely mentioning or quantifying it.

Table 4.1: Related work concerning identification of virtualization overhead.

Study	Publication year	Guest overhead	Host overhead	OC overhead
[46]	2006		X	
[139]	2007			X
[110]	2008			X
[107]	2008	X	X	
[137]	2008	X		
[133]	2010	X		
[142]	2010	X	X	X
[116]	2011		X	
[136]	2011	X	X	
[53]	2011	X	X	
[160]	2011		X	X
[144]	2011	X		
[16]	2012		X	
[22]	2012	X	X	
[114]	2013			X
[50]	2013		X	X
[108]	2013	X		
[113]	2013			X
[161]	2013		X	
[14]	2014		X	
[118]	2014	X		
[162]	2014		X	X
[10]	2015	X	X	X
[163]	2015		X	
[13]	2016			X
[55]	2016	X	X	
[7]	2016		X	
[117]	2016			X
[164]	2016			X
[165]	2016	X		
[138]	2016			X
[109]	2017			X
[153]	2017	X		
[9]	2018		X	
[166]	2018			X
[103]	2019			X
[154]	2019	X		
[167]	2020	X	X	
[168]	2020	X		
[56]	2021	X	X	

Table 4.1 indicates that indeed many existing studies address at least one of the categories of virtualization overhead described in this chapter. Some of them even do so in great detail. However, it is also clear that this detail only extends to a specific aspect of the overhead. Table 4.1 lists only two studies ([142] and [10]) that address at least some aspect of all three categories. Of these two, only [10] does so in a systematic manner comparable to this chapter. Since this study was published in 2015 however, it can no longer be considered representative for modern virtualized systems since virtualization technology has evolved so profoundly in the past decade. Therefore, table 4.1 makes clear that this work is the only effort to provide a clear and all-encompassing picture of virtualization overhead suffered by multithreaded applications on modern systems.

4.5 Conclusion

Thanks to persistent efforts from academia and industry, contemporary hardware-assisted x86 virtualization techniques induce minimal overhead for sequential computation-intensive workloads on modern platforms. Unfortunately, this is not yet the case for their multithreaded counterparts. Overhead may have many different causes which each manifest themselves in a unique way depending on the workload and system configuration. The perceived application effects may differ greatly from the underlying impact on the system. The relationship between these system and application effects is primarily determined by the critical path of the workload. The principal remaining causes of virtualization overhead for multithreaded applications are thread-coordination and NUMA management.

While this chapter has touched on many known issues, the enormous advances in virtualization technology in the last decade have rendered almost all existing work regarding this topic outdated. Especially considering that this chapter uncovered several as of yet unknown causes of virtualization overhead for the target workloads of this dissertation, it is in the estimation of the author a valuable contribution to the field as well as an adequate answer to the first pair of secondary research questions established in §1.3.

4.5.1 Personal Contribution

In addition to the main author of this dissertation, several parties were involved with the work presented in this chapter through providing the test platform used, performing several of the required experiments and collaborating on interpreting findings. While it is thus unfair to state that any part of this chapter is the exclusive contribution of this dissertation's author, he did have a principal role throughout and was in the end responsible for concatenating individual data points to a cohesive narrative.

Chapter 5

Reducing Virtualization Overhead for Multithreaded Applications

This chapter was previously published as part of:
S. Schildermans et al. "Virtualization Overhead of Multithreading in X86 State-of-the-Art & Remaining Challenges". In: *IEEE Transactions on Parallel and Distributed Systems* 32.10 (2021), pp. 2557–2570

The previous chapter has made clear that multithreading still induces substantial virtualization overhead. While this overhead stems from a multitude of sources, it can be conceptually grouped in the following categories:

- **Blocking synchronization**: Blocking-based primitives designed to coordinate the execution flow of a multithreaded application;
- **Spinning synchronization**: Spinning-based primitives designed to coordinate the execution flow of a multithreaded application;
- **Data sharing**: Operations induced by threads modifying shared data;
- **NUMA opacity**: Issues induced by abstraction of the host NUMA architecture.

For each of the above categories, this chapter discusses the most common techniques employed today to overcome their inherent virtualization overhead. While these techniques have already briefly been mentioned in previous chapters to facilitate interpretation of the results presented there, this chapter provides a much more detailed analysis thereof in order to understand their impact on virtualized workloads in greater depth. Moreover, this chapter presents and discusses a range of novel approaches to further reduce virtualization overhead for multithreaded applications. Some of these have already been proposed in literature, while others are original ideas.

Main Findings & Contributions

- While halt polling improves $\delta\eta_t$ for blocking synchronization, it greatly increases $\delta\eta_r$;
- Alternative techniques to reduce the cost of vCPU scheduling are under development, but not yet mature;
- While hardware assistance has greatly optimized virtualizing IPIs, strict co-scheduling is the only known method to further improve this mechanism. However, this technique has known resource fragmentation issues;
- Tweaking the scheduler tick behavior may reduce virtualization overhead related to blocking synchronization for specific workloads;
- Paravirtualizing the scheduler tick has the potential to significantly reduce virtualization overhead for blocking synchronization;
- Exploiting symmetric multithreading (SMT) may drastically reduce virtualization overhead related to scheduling and NUMA opacity;
- While PLE is highly effective at mitigating spinning synchronization overhead at kernel level, it currently does not address user-level spinning;
- Spin-then-block primitives offer a good alternative to traditional spin locks to minimize spinning at both user and kernel level;
- Compilers can be enhanced to detect user-level spinning constructs and replace them by virtualization-friendly alternatives;
- Pause exiting may provide a fundamental solution to the issue of excessive spinning in virtualized systems, albeit while degrading spin lock performance in some cases;
- System calls implementing spinning synchronization would allow applications to utilize PLE at a limited cost in spin lock performance;

- Alternative TLB designs may eliminate the need for TLB shootdowns and their associated virtualization overhead;
- While application source code alteration may be effective at reducing TLB shootdowns, altering memory allocator behavior is a much more programmer-friendly approach;
- Extended paravirtualization may eliminate the NUMA opacity problem without constraining the potential for resource consolidation;
- Modern techniques to optimize vCPU placement are still lacking;
- In general, application-level solutions to reduce virtualization overhead are highly promising but understudied as of now.

5.1 Blocking Synchronization

§4.2.3 has demonstrated that blocking synchronization is a complex affair in virtualized systems, inducing up to four VM exits for every synchronization cycle when the lock in question is heavily contended. The same section describes that multiple distinct causes underlie these VM exits. Said causes are best treated as independent issues with dito potential solutions. Consequently, several research directions as well as industrial innovations benefit blocking synchronization in a virtualized context. Below an elaboration on each of the existing innovations known to the author, supplemented with original suggestions to further reduce the virtualization overhead associated with this synchronization mechanism.

5.1.1 Deferred Scheduling

The most extensively studied aspect of virtualization overhead related to blocking synchronization is reducing the cost of vCPU scheduling. The best example of such efforts is the concept of halt polling, which has already been adopted by some VMMs (e.g. KVM). While—as clarified in §4.2.3—this technique may reduce $\delta\eta_t$ related to vCPU scheduling (which is often induced by blocking synchronization), that section equally suggests that halt polling itself may have a non-negligible negative impact on $\delta\eta_r$.

To clarify the above perception, figure 5.1 compares $\delta\eta_r$ and $\delta\eta_t$ for the experiments performed in §4.2.3 with halt polling respectively enabled and disabled. Only the UC scenario is considered, since halt polling has a negligible impact on performance in the OC scenario (as also explained in §4.2.3).

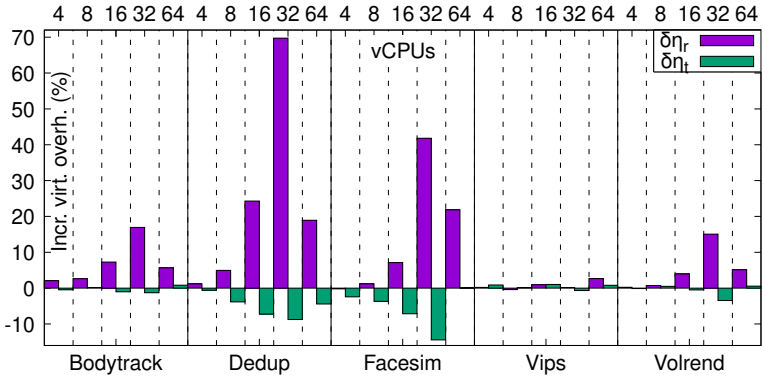


Figure 5.1: $\delta\eta_r$ and $\delta\eta_t$ caused by halt polling for the benchmarks with high host overhead per vCPU count in the UC scenario.

Figure 5.1 confirms the conjecture that halt polling is not at all resource-efficient. While it does reduce $\delta\eta_t$ by up to 14%, this comes at a great cost in $\delta\eta_r$. When raw application performance is the only concern, this is justifiable. However, these days this philosophy is highly debatable for various reasons, not least the tendency of cloud providers to charge consumers at ever-finer granularities, down to milliseconds of CPU time [83]. This means that an increase in $\delta\eta_r$ is directly charged to the consumer, making totally disregarding $\delta\eta_r$ in favor of $\delta\eta_t$ an ever more dubious system design choice. Besides these efficiency concerns, as already stated in §4.2.3, halt polling is hardly effective to begin with when the system is overcommitted and/or VM vCPU counts are large, indicating that it is not a durable solution since cloud environments tend to be heavily consolidated and VM vCPU counts continue to increase [150].

The above issues are inherent to the polling concept. It is very hard to balance performance and efficiency, especially on overcommitted systems where any cycles spent on polling reduce system throughput. The reluctance of Xen to adopt any form of halt polling underpins this. Therefore more intelligent solutions are highly desirable. Existing research has attempted to replace polling by computation migrated from other vCPUs, but this introduces vCPU overloading as a side effect [14]. A recent solution, [164], can reduce such side effects but requires substantial changes to the guest OS, which limits its potential for rapid and widespread adoption. All of this suggests that deferred vCPU scheduling is to be viewed as a specialist tool to tune VM performance rather than as a silver bullet improving vCPU scheduling behavior in general terms.

5.1.2 Interrupt Controller Virtualization

Handling IPIs—and interrupts in general—efficiently in a virtualized environment has received much attention from hardware manufacturers. Intel's APICv and AMD's AVIC reduce IPI-induced virtualization overhead by approximately 60% by managing interrupt delivery and acknowledgement in hardware [16]. Nevertheless, the results presented in §4.2.3 indicate that this issue is still significant. Specifically the `RESCHEDULE` IPIs associated with blocking synchronization are of critical importance to application performance, since the thread being awoken may only resume execution upon receipt of the IPI. Given that blocking synchronization is by definition a serializing construct, it is likely that many of these `RESCHEDULE` IPIs are part of the application's critical path. As such, ω is high for this particular form of virtualization overhead, meaning that reducing it is likely to have a significant positive effect on $\delta\eta_t$, even if $\delta\eta_r$ is only modestly ameliorated. Further improvements in this regard are therefore highly desirable.

5.1.3 Co-Scheduling

Beyond the already adopted hardware improvements mentioned above, strict co-scheduling has been proposed to eliminate the need for intercepting IPIs in a virtualized environment because whenever a guest CPU sends an IPI, the receiving vCPU would be guaranteed to be active. Existing hardware assistance for interrupt rerouting (see §2.2.4) may be employed to map vCPU identifiers (IDs) to corresponding pCPU IDs. However, the major drawback of strict co-scheduling is CPU fragmentation [50]. Namely, co-scheduling demands that all of a guest's vCPUs are scheduled and descheduled simultaneously, which means that a VM with eight vCPUs performing a sequential workload occupies eight pCPUs at all times, even when it is sharing the system resources with other VMs which would be able to utilize the occupied resources much more efficiently. Moreover, if no combination of VMs can be found so that the sum of the vCPUs used by those VMs equals the number of available pCPUs, some system resources will inevitably be continuously idle. It is therefore clear that alternative solutions are direly needed. To the best knowledge of the author, this remains an open question to date.

5.1.4 Scheduler Tick Management

One of the most interesting findings in §4.2.3 is the fact that the VM exits induced by guest scheduler tick management account for a $\delta\eta_r$ of up to 10% for applications relying heavily on blocking synchronization. The most surprising aspect of this issue is that it is a direct consequence of tickless kernel operation (see §4.2.3), which has been described in literature as having an exclusively positive effect on virtualization compared to traditional periodic ticks [156]. Since this issue has not even been acknowledged in existing literature, it is self-evident that no explicit mitigation techniques exist.

Despite lacking explicit mitigation techniques, intelligent system configuration may work around the problem described above. Specifically, the Linux kernel allows for tweaking the behavior of the scheduler tick through the boot parameter `CONFIG_NO_HZ` [155]. One may choose to never disable the scheduler tick (referred to as classic periodic ticks), only disable it on idling CPUs (tickless kernel mode, a.k.a. dynticks idle mode), or disabling it on CPUs that have at most one runnable task available (full dynticks mode). However, while reverting to classic periodic ticks may eliminate excessive virtualization overhead for applications relying heavily on blocking synchronization, it obviously reintroduces the virtualization issues with classic periodic ticks described in literature. Namely, the VMM must handle each vCPU's tick interrupts individually. A heavily overcommitted host may therefore spend a significant amount of its resources on handling tick interrupts for idle vCPUs, which leads to massive virtualization overhead [156]. Knowing this, full dynticks mode at first glance seems to be an ideal solution, since it eliminates the need to disable the tick upon every transition between idle and active vCPU states while at the same time not requiring tick interrupts for idle vCPUs. However, this only holds true for specific workloads. Namely, this approach simply shifts the threshold for disabling the scheduler tick on a particular vCPU from having no runnable tasks to having one runnable task. As such, multithreaded workloads that are not specifically tuned to employ exactly one worker thread for each available vCPU may experience just as much or even more virtualization overhead related to scheduler tick management using full dynticks mode as they would using dynticks idle mode. Therefore, tuning the scheduler tick is a specialist tool rather than an all-round solution to the problem of virtualization overhead induced by scheduler tick management.

Because this work is to the knowledge of the author the first to expose the issues associated with tickless kernel operation in virtualized systems, it is a natural reflex to also be the first to provide a solution to said issues. In light of this, this dissertation presents the concept of virtual scheduler ticks. This idea completely reconsiders how scheduler ticks are managed in a virtualized environment. After

all, the scheduler tick is in essence a mechanism to tie the system's notion of the passing of time to physical time through interaction with hardware devices. In a bare metal context, this is unquestionably a responsibility of the OS. In a virtualized environment on the other hand, the VMM acts as the OS with regard to hardware management, essentially taking over this duty from the guest kernels. Because guest kernels are normally not aware of the fact that they are being virtualized and thus do not voluntarily yield this responsibility to the VMM, the latter must forcefully intercept any guest attempt to alter the timer hardware, which introduces the tick-related virtualization overhead described in §4.2.3. From a conceptual standpoint, it would be far more prudent if the VM would proactively delegate management of the scheduler tick to the VMM. In essence, a guest should be able to request scheduler ticks from the hypervisor much like applications may request system services from the OS. The VMM would then be responsible for performing the necessary hardware interactions to provide this service. This is the basic idea behind virtual scheduler ticks. Chapter 6 is dedicated to the refinement, implementation and evaluation of this concept.

5.1.5 Symmetric Multithreading

One may argue that in essence, all of the issues with blocking synchronization in a virtualized setting are caused by discontinuous CPU availability to (idle) vCPUs. Following this logic, virtualization overhead related to scheduling—and thus blocking synchronization—may be drastically reduced by ensuring a vCPU is never fully descheduled. Obviously, this stands in direct contrast to one of the principal goals of virtualization, being hardware consolidation. However, these conflicting goals may be reconciled by exploiting the SMT capability of many modern CPUs. Recent work applies this idea through statically assigning a dedicated SMT context to each vCPU, spreading all vCPUs of a particular VM over distinct pCPUs, but allowing vCPUs from distinct VMs to occupy distinct SMT contexts within a particular pCPU [169]. In this way, there is no need to deschedule vCPUs at all while in most cases not significantly reducing system throughput, thus greatly reducing scheduling-related virtualization overhead without considerable side effects. The main drawback of this technique however is that it requires highly capable hardware. Concretely, the host must sport at least as many pCPUs as the number of vCPUs of the largest VM to be hosted and at least as many SMT contexts per pCPU as the number of VMs to be hosted simultaneously. While at the moment these constraints can be considered too stringent from a pragmatic perspective, it is reasonable to assume that this approach will be viable in the foreseeable future, since many-core CPUs containing eight SMT contexts per core already exist [170].

5.1.6 Synchronization-Aware Application Design

While the above has made clear that further refinements to the virtualization process still have plenty of potential to reduce virtualization overhead related to blocking synchronization, system-level solutions will always have to consider certain design trade-offs to ensure correctness and efficient execution of all workloads they may encounter, which tends to impose restrictions on the performance gain that may be achieved. Moreover, widespread adoption of novel mitigation techniques at system level is more often than not a slow process which may easily take years to make a considerable impact in the real world. For these reasons, conscientious application developers may instead consider tackling virtualization overhead in a direct manner, namely through purposely designing their applications in such a way that they make minimal use of operations which may induce excessive virtualization overhead. To the surprise of the author, this approach has received little to no attention in literature. As such, this section aims to provide an indication of the potential of this concept.

Intuitively, an effective way to reduce thread-interdependencies and thus the need for (blocking) synchronization is focusing on data parallelism during the application design process. Therefore, this principle is an ideal candidate to assess the effectiveness of intelligent application design as a means to mitigate virtualization overhead. Equally intuitively however, adopting any such a fundamental design principle may be far from trivial in some cases. Besides imposing restrictions on the application architect's freedom, such an endeavor may in the case of existing applications require rewriting large amounts of source code. These days however, solutions aiding in this process exist. For example, nowadays many programming languages provide libraries allowing developers to implement common parallel design patterns with minimal effort by abstracting implementation details such as thread creation and synchronization from developers. Danelutto et. al. have employed one such library to implement the PARSEC benchmark suite in a data-parallel manner [171]. We profiled their implementation to asses its effectiveness in reducing virtualization-sensitive synchronization operations. Figure 5.2 shows the results for all the PARSEC benchmarks identified in §4.2.3 as exhibiting high blocking synchronization-related virtualization overhead, broken down per vCPU count in the UC scenario.

Figure 5.2 shows promising results. All synchronization operations have been reduced by up to 70%. This improvement tends to increase with vCPU count. One exception seems to be the `HLT` operations induced by the *Dedup* benchmark. However, profiling *Dedup* in detail reveals that these operations are induced by I/O rather than synchronization. Figure 5.2 also suggests this, as the `RESCHEDULE` IPIs are drastically reduced. Thus, it is safe to conclude that intelligent application design may indeed help considerably in reducing

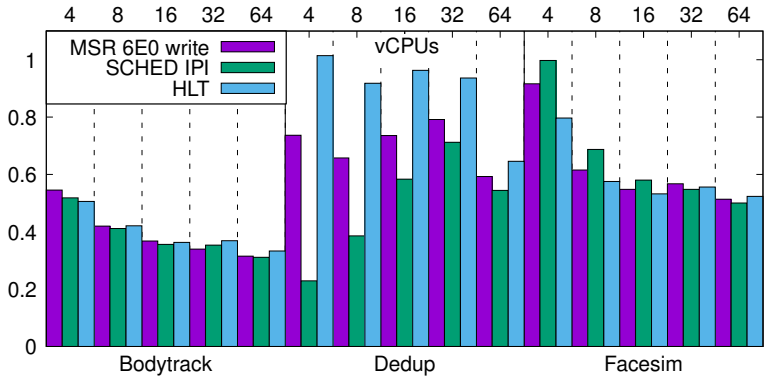


Figure 5.2: Number of virtualization-sensitive synchronization operations for the P3ARSEC workloads relative to their original equivalents that show many such operations per vCPU count in the UC scenario.

virtualization overhead related to blocking synchronization. Because of these promising results, it would be negligent not to explore this trajectory further in this dissertation. As such, chapter 8 explores mitigating virtualization overhead at application level in much greater depth.

5.2 Spinning Synchronization

As stated in §3.2.5, spinning synchronization may induce exorbitant amounts of virtualization overhead when the host is overcommitted. Chapter 4 identified two forms of spinning synchronization: spinning at kernel level on the one hand and user-level spinning on the other. While §4.2.3 indicates that virtualization overhead induced by the former has been mostly mitigated through recent enhancements to the virtualization process, the latter remains a severe issue. Therefore, novel approaches to deal with (user-level) spinning in virtualized settings are direly needed. This section proposes several such novel approaches and elaborates on the existing techniques that have proven effective at mitigating kernel-level spinning, exposing their limitations and suggesting further refinements.

5.2.1 Pause Loop Exiting

The findings discussed in §4.2.3 show that considerable progress has been made in dealing with LHP and related issues in recent years. For example, only half a decade ago, $\delta\eta_t$ was over 500% for the *Dedup* benchmark in OC settings [10]. Figure 4.4 shows that thanks to modern system enhancements, $\delta\eta_t \approx 20\%$ (OC) or $\delta\eta_t \approx 50\%$ (OC_2) for the same workload nowadays.

While PLE has proven effective at dealing with excessive spinning at kernel level, one of its main limitations is that it only functions when the CPU is operating in kernel mode. This means that it cannot be used to address user-level spinning in its current form. It is unclear to the author why PLE was designed this way. Extensive literature review has not revealed any reasoning for this design decision. The most likely explanation for this observation is that hardware manufacturers do not wish to interfere with the behavior of (often carefully implemented) application synchronization protocols. While to expert application developers this is certainly an advantage, to all others this decision may lead to unintended grave performance degradation. Therefore, the author argues that PLE should be available at application level. Concerns about interfering with application behavior may be addressed by exposing a PLE configuration MSR to user space, allowing expert users to disable this function if they so desire.

Despite the reassuring results presented in §4.2.3, even with the enhancements suggested above PLE is not a fundamental solution to the problem of excessive spinning in overcommitted virtualized systems because it may still allow for a significant amount of spinning to take place before intervening, as noted in §4.2.4. As such, mitigating LHP is still to be considered an ongoing issue, with PLE representing a significant step in the right direction.

5.2.2 Paravirtualized Ticket Spin Locks

As mentioned in §3.2.5, another existing technique to address both LHP and LWP is the adoption of paravirtualized ticket spin locks. While such locks have certainly proven effective at reducing spinning-related virtualization overhead [114], much like PLE, they rely on spinning detection at runtime and can therefore eliminate all futile spinning induced by LHP and LWP. Moreover, their reliance on paravirtualization hinders their widespread adoption. Currently, these locks are—to the best knowledge of the author—only available to Linux guests running on Xen or KVM hosts [113]. Thus, like PLE, paravirtualized ticket spin locks are best viewed as a pragmatic intermediary solution pending an effective, more fundamental alternative.

5.2.3 Pause Exiting

A simple method to avoid the inefficiency related to ad-hoc detection of spinning as it is already occurring upon which both existing methods to mitigate virtualization overhead related to spinning synchronization mentioned above rely is to employ 'pause exiting' rather than pause-loop exiting. This is a capability already present in modern x86 CPUs, which—if enabled—generates a VM exit on each `PAUSE` instruction [42]. On such an exit, the VMM may schedule a different vCPU if the system is heavily overcommitted or reschedule the vCPU that generated the VM exit immediately if not, until a threshold is reached. If the vCPU keeps exiting, LHP is likely and the exiting vCPU can be descheduled for a longer time. Intelligent algorithms may be developed to determine the amount of time between attempts at rescheduling the exiting vCPU in function of the amount of contention. This principle is in fact similar to halt polling. Note that this technique may easily address both spinning at kernel and user level, since—as opposed to PLE—hardware support for pause exiting is already available in both user and kernel space.

While pause exiting may improve performance by minimizing spinning in the event of LHP or LWP, the cost of repeated VM exits may largely mitigate potential performance gains, in particular for highly contended locks protecting short critical sections. On the other hand, this is a fundamental solution to LHP/LWP which does not burden application developers and does not require novel hardware extensions. Therefore, in the opinion of the author this idea warrants further investigation.

5.2.4 Blocking Synchronization

Even though §5.1 highlighted plenty of issues concerning blocking synchronization in a virtualized context, §4.2.4 has shown that these issues are limited compared to the potential performance impact of user-level spinning. Therefore, replacing any user-level spinning synchronization primitives by blocking-based ones in application source code may be a sensible approach to drastically reduce overall virtualization overhead. We explored this idea for the *Volrend* benchmark, which was identified in §4.2.4 as suffering most from user-level spinning and found that $\delta\eta_r$ and $\delta\eta_t$ were reduced by resp. 60% and 25% in the OC scenario with 64 vCPUs. Given the magnitude of this improvement, it is reasonable to conclude that this approach is indeed a viable method to reduce virtualization overhead induced by user-level spinning in the general sense.

Plainly replacing spinning by blocking synchronization may not fit the needs of all applications because of the naturally lower performance of blocking synchronization (irrespective of the effects of virtualization). Combined with the issues blocking synchronization itself induces in a virtualized context, many applications are likely better served by a hybrid spin-then-block synchronization mechanism. Many programming languages provide ready-to-use implementations of such primitives (e.g. `InitializeCriticalSectionAndSpinCount` in C++ [172]) or even implement the spin-then-block mechanism directly in the language runtime environment, completely abstracting its implementation from application developers (e.g. Oracle's JRockit JVM [173]). For languages lacking such a feature, programmers may design custom primitives implementing this principle. Such primitives are likely to constitute an ideal balance between the risks of user-level spinning and the performance penalty of blocking synchronization for many applications.

5.2.5 Compiler Enhancements

Naive user-level spin lock implementations tend to exhibit a similar, simple structure akin to the pseudocode shown in listing 5.1. It is feasible for compilers to identify such structures and replace them with more virtualization-friendly alternatives. This could either involve replacing these constructs by spin-then-block primitives or injecting `PAUSE` instructions within the loop. Note that the latter would require PLE to be supported at user-level as well in order to significantly reduce excessive spinning. Notwithstanding, injecting `PAUSE` instructions in any spinning-based synchronization primitive is highly desirable, even in native scenarios. Namely, this instruction was specifically designed to notify the CPU that the application is waiting for a spin lock in order to avoid memory order violations, which drastically improves spin lock performance on modern CPUs with advanced branch prediction [174].

```
global int lock;
...
while (!atomic_compare_and_swap(&lock,0,1));
...
lock = 0;
```

Listing 5.1: Strucure of a generic user-level spin lock.

5.2.6 Spin Lock System Calls

At the heart of the user-level spinning issue lies the fact that currently, OSs do not expose their internal spinning synchronization primitives to applications [115]. This obligates application (runtime) developers wishing to implement spinning synchronization to come up with their own interpretation of the concept. Even without considering virtualization, it is evident that many of these ad-hoc user-level spin locks are not implemented in an optimal way (e.g. not employing the `PAUSE` instruction). By simply exposing the well-defined spinning primitives employed by the OS to applications through the system call interface, application developers would no longer need to implement their own—likely sub-optimal—versions of this mechanism. Moreover, this approach would greatly reduce virtualization overhead related to user-level spinning, because the actual spinning would take place at kernel level, allowing PLE to intervene when LHP or LWP occur.

The obvious drawback of offering spinning synchronization as an OS service through the system call interface is that it requires source code alterations to make it available to existing applications. More worryingly however, the overhead involved in invoking a system call and switching to kernel space may defeat the main purpose of spinning synchronization—avoiding the context switch overhead related to blocking synchronization—in the first place. However, system calls are still much less costly than full context switches. Additionally, spin lock system calls may be implemented in a hybrid manner, where much of their code is executed in user space and the switch to kernel space is only made when absolutely necessary (i.e. when the lock is contended). The main system call upon which blocking synchronization is based in Linux—`futex`—employs exactly this strategy as well [175].

5.2.7 Co-Scheduling

Much like with blocking synchronization, co-scheduling may entirely eliminate the issues associated with spinning synchronization in a virtualized context, since it forbids vCPUs holding or waiting for a spin lock to be descheduled while other vCPUs from the same VM may be attempting to acquire it. However, this technique comes with its own limitations, as outlined in §5.1.3.

5.3 Data Sharing

Sharing data between threads running concurrently on distinct vCPUs may induce significant virtualization overhead through TLB consistency management, as §3.2.7 describes. While §4.2 has indicated that addressing this issue is pressing, it has received much less attention from literature than the issues previously discussed in this chapter. Therefore, this section proposes several potential techniques

Below several improvements to TLB design and the TLB shootdown process which have the potential to drastically reduce virtualization overhead related to inter-thread data sharing are proposed.

5.3.1 Interrupt Controller Virtualization

Since TLB shootdowns are implemented using IPIs, both APICv and AVIC benefit them as much as they benefit `RESCHEDULE` IPIs in the context of blocking synchronization, as discussed in §5.1.2. However, as previously discussed this mitigation technique does not eliminate all VM exits related to sending IPIs. Moreover, it does not address TLB shootdown preemption. As such, this hardware-level enhancement must be supplemented by other techniques in order to sufficiently address virtualization overhead induced by TLB shootdowns.

5.3.2 Alternative Translation Lookaside Buffer Design

Beyond reducing virtualization overhead associated with TLB shootdowns, one may attempt to eliminate the need for them in the first place. To that end, many alternative TLB designs have been proposed [176]:

- **Shared TLB**: Some work proposes to implement the TLB as a shared cache. While this approach obviously eliminates the need for TLB consistency enforcement, the main challenge with this approach is performance. Namely, modern x86 CPUs employ a virtually indexed, physically tagged (VIPT) cache structure, meaning that cache lookup may only complete once the TLB returns a result [42];
- **Hardware-Managed TLB consistency**: Various methods have been proposed to implement TLB consistency in hardware. In fact, it is not entirely clear why this is not yet the default approach in x86. Cited reasons for this include reliability and performance, but strangely the main driver seems to be tradition [177].

Several prototypes exist of the proposed alternative TLB architectures described above. These architectures can be easily extended to work for virtualized systems since most contemporary TLBs already contain a VM ID tag for each TLB entry, eliminating the need for TLBs to be flushed upon VM exits/entries and thus allowing the TLB to operate identically in respectively a native or virtualized environment [42]. As of now, there are however no plans known to the author to adopt said alternative TLB designs on a large scale. It will therefore take at least several more years for any of these designs to have a meaningful impact on virtualization overhead, since hardware improvements only slowly trickle down to industry due to the investments involved.

5.3.3 Co-Scheduling

Analogously to blocking and spinning synchronization (see §5.1.3 and §5.2.7, respectively), strict co-scheduling may eliminate the need for the VMM to handle TLB shootdown IPIs as well as TLB shootdown preemption through enforcing all vCPUs associated with a particular VM to be scheduled simultaneously. Refer to §5.1.3 for a detailed description of this technique and its drawbacks.

5.3.4 Source Code Alteration

In §4.2.3 the high-level cause of most TLB shootdowns for multithreaded applications has been identified as heap resizing. Since this heap resizing is a direct consequence of the application allocating or releasing memory, it is evident that the amount of TLB shootdowns induced by the application may be drastically reduced by changing its memory allocation behavior at source code level. Like co-scheduling, source code alteration has been proposed in the context of blocking synchronization (§5.1.6) and spinning synchronization (§5.2.4) as well. However, regarding minimizing heap resizing this approach is particularly challenging since modern memory allocators are very complex. Identification and amelioration of problematic code without greatly compromising memory efficiency requires a deep understanding of the particular memory allocator used and is therefore highly challenging. Nonetheless, chapter 8 provides several guidelines that aid application developers in precisely this effort.

5.3.5 Alternative Memory Allocator Design

Rather than requiring application developers to alter their source code as suggested above, the number of TLB shootdowns applications induce may also be drastically reduced by altering the memory allocators used by these applications so that they call system routines performing said TLB shootdowns as little as possible. This will however come at the inevitable expense of some memory efficiency, since balancing application memory efficiency and costly interaction with the system in order to allocate or release memory is intuitively a fundamental trade-off in memory allocator design. However, relevant literature does not ever seem to consider this trade-off explicitly. Rather, the main trade-off under consideration is relieving thread contention (favored by high-performance allocators such as tcmalloc [178]) versus maximizing memory efficiency (favored by high-efficiency allocators such as ptmalloc2 [179]). Any allocators exhibiting low TLB shootdown overhead therefore achieve this as a side effect of other design decisions rather than as an explicit design goal.

In spite—or perhaps because—of the lack of attention TLB shootdowns have received from memory allocator developers, §4.2.3 indicates that it is high time to start considering the role TLB shootdowns play in application performance from a memory allocator design perspective. This issue will likely become even more pressing towards the future, given the ever-increasing emphasis on virtualization on the one hand and parallelism on the other in industry [2, 150]. This dissertation provides a first step in the right direction regarding this challenge by developing a novel memory allocator design concept named 'global hysteresis'. This concept balances memory efficiency and TLB shootdowns better than any existing memory allocator design paradigm known to the author. Chapter 7 elaborates on global hysteresis and describes a prototype implementation thereof based on ptmalloc2.

5.4 Non-Uniform Memory Access Locality

The final high-level cause of virtualization overhead for multithreaded applications identified in chapter 4 is the opacity of the physical system's memory layout to the VM. This issue may drastically increase memory latency as a consequence of improper scheduling decisions on the guest's part if the host system sports a NUMA architecture. Several approaches already exist to deal with this issue. Two methods are common, as alluded to in §3.2.8: NUMA passthrough and dedicated NUMA locality managers. This section discusses both of these approaches in detail, in addition to some less orthodox novel techniques.

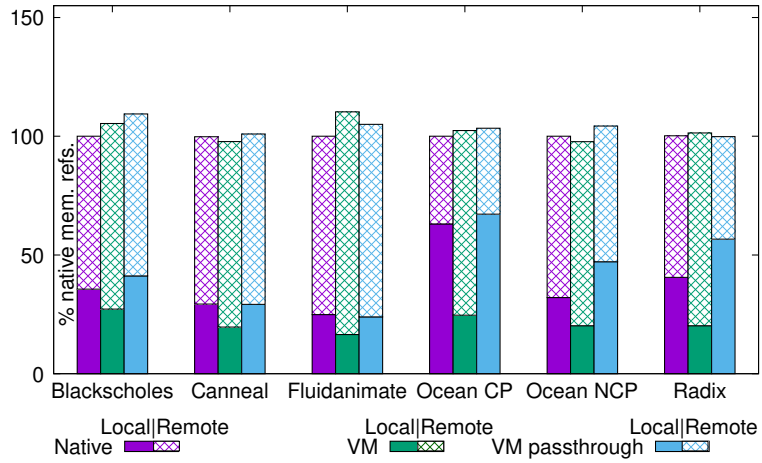


Figure 5.3: Memory locality of NUMA passthrough for the benchmarks studied in §4.2.2 in the UC, 64 vCPU scenario, normalized to native.

5.4.1 Non-Uniform Memory Access Passthrough

The most straightforward method to address the NUMA opacity issue is to pass through the NUMA architecture of the host system to the VM. This involves pinning each vCPU to a set of pCPUs belonging to a singular host NUMA node and presenting the guest with a virtual NUMA architecture constructed so that all vCPUs pinned to a particular host NUMA node belong to the same virtual NUMA node. This allows the guest scheduler to optimize scheduling decisions with regard to the virtual NUMA architecture of the VM, which by proxy is the physical NUMA architecture of the host. Every major VMM offers this ability [154], which in principle yields VM memory latency identical to that of the physical system represented by that VM. Figure 5.3 assesses this for the benchmarks identified in §4.2.2 as suffering from the NUMA opacity issue by comparing the number of local and remote memory accesses performed by these benchmarks in a native setting, a VM without optimizations and a VM employing NUMA passthrough. These results were collected using `pcm-numa`[1].

The results presented in figure 5.3 are in line with expectations. Firstly, memory locality is greatly reduced for all benchmarks when run in a VM without optimizations. Secondly, manual NUMA exposure mitigates this issue entirely. This technique is thus certainly a viable option to improve performance for virtualized workloads exhibiting excessive memory latency.

[1] `https://github.com/opcm/pcm`

While NUMA passthrough does achieve its principal goal, it comes with several undesirable side effects. Most importantly, it reduces the potential for resource consolidation, since vCPUs can no longer be migrated between NUMA nodes without compromising the advantages of NUMA passthrough. Moreover, constructing virtual NUMA layouts can be tedious, especially for large VMs. Lastly, VMs employing this technique can no longer easily be migrated between hosts with different NUMA configurations. Essentially, NUMA passthrough thus achieves performance gains through sacrificing some of the flexibility the virtualization process offers. Therefore it is not applicable in all circumstances and its utility must be considered on a case-by-case basis.

5.4.2 Non-Uniform Memory Access Locality Managers

Another commonly used approach to combat the NUMA opacity issue is taking the host NUMA architecture into account at VMM level, in particular when scheduling vCPUs. This technique may be implemented directly in the VMM scheduler or in a dedicated utility program that runs alongside the VMM, advising it on optimal vCPU placement in real time. Many algorithms have been developed in this regard, as refinement of this technique is to date the subject of active research [153, 154]. Within the context of Linux/KVM, a popular example of such an algorithm is implemented in the form of `numad`, which is a dedicated NUMA locality management daemon or KVM[2]. Figure 5.4 shows how this algorithm performs in experiments analogous to those presented in figure 5.3. While other algorithms may yield varying performance, it is reasonable to assume figure 5.4 provides some insight in their general behavior.

Surprisingly, figure 5.4 suggests that `numad` outperforms native execution in terms of memory locality. On the other hand, its total performance impact on the system seems to be unpredictable, given that for some benchmarks, the total number of memory accesses performed by the system employing `numad` far exceeds that of an equivalent system not doing so. To verify this intuition, we analyzed how `numad` impacts $\delta\eta_r$ for the benchmarks shown in figure 5.4. Figure 5.5 shows the results.

Figure 5.5 reveals that for most benchmarks results are in line with expectations. Note that a small amount of residual $\delta\eta_r$ is to be expected when using `numad` due to the resource consumption of `numad` itself on the one hand and virtualization overhead not related to the NUMA opacity issue on the other. This does however not hold true universally, as indicated by the results for the benchmarks *Canneal* and *Ocean CP*. Regarding the former, `numad` seems to outperform even native execution. This is unlikely, but possible given that any scheduler employs a set

[2] `https://linux.die.net/man/8/numad`

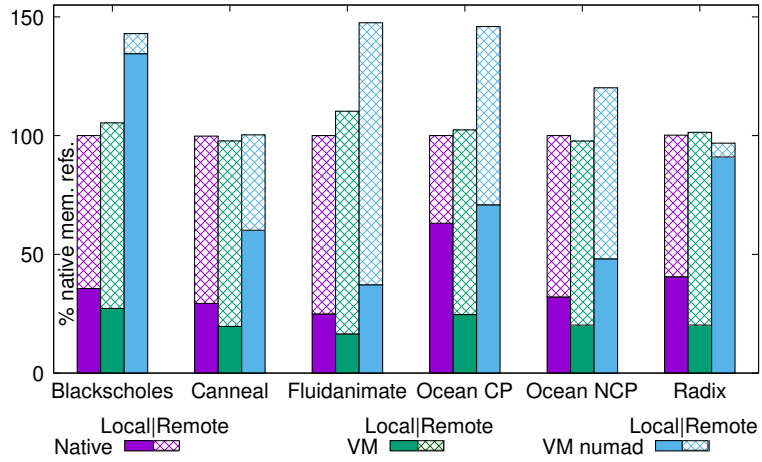


Figure 5.4: Memory locality of `numad` for the benchmarks studied in §4.2.2 in the UC, 64 vCPU scenario, normalized to native.

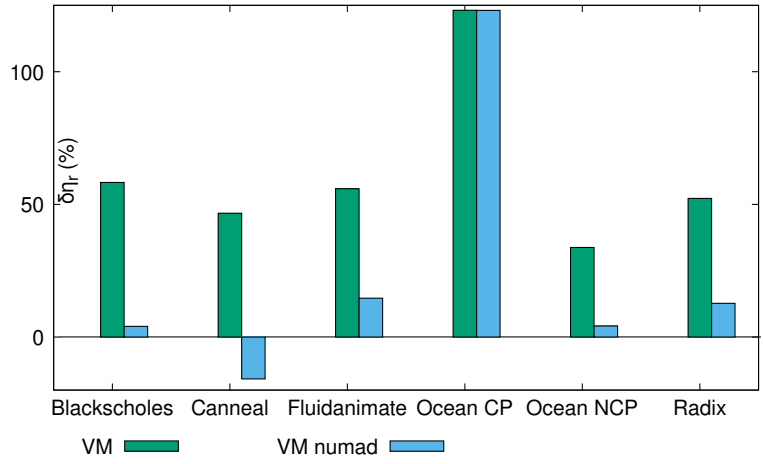


Figure 5.5: $\delta\eta_r$ for the benchmarks studied in §4.2.2 in the UC, 64 vCPU scenario for a system employing `numad` compared to a system without dedicated NUMA management.

of heuristics to determine NUMA placement. Therefore, any NUMA placement strategy is likely to perform excellently for some workloads and poorly for others. *Canneal* appears to clearly favor the heuristics employed by `numad`, while for other workloads this is not the case; or even the opposite is true. Speaking of which, *Ocean CP* consumes just as many system resources when run on a system that employs `numad` as it would when run on a system that does not. This may be explained by the fact that as noted in §4.2.2, this benchmark is bottlenecked by memory bandwidth. Therefore, improving memory locality may in fact be counterproductive in this specific case, as this likely leads to data being spread over fewer NUMA nodes, reducing the total available memory bandwidth. While more research into this phenomenon is needed to assess how memory locality managers other than `numad` behave in this scenario, it is clear that employing a NUMA locality manager does not by definition translate into improved memory performance. Analogously to NUMA passthrough, NUMA managers are thus to be seen as a tool that may be employed by advanced users in order to improve performance for specific workloads rather than a general solution to the issue of NUMA opacity.

5.4.3 Symmetric Multithreading

Sections 5.4.1 and 5.4.2 have made clear that neither of the mainstream existing approaches to combat NUMA opacity in VMs perform satisfactorily across workloads. Therefore, it is prudent to consider alternative approaches to deal with this issue. One such approach has already been discussed in §5.1.5 in the context of blocking synchronization, namely pinning vCPUs to dedicated SMT contexts in order to eliminate the need for most to all vCPU scheduling. This technique has the potential to eliminate the NUMA opacity issue as well because it guarantees that any particular vCPU is pinned to a specific pCPU and therefore NUMA node. This allows for the physical NUMA architecture to be automatically exposed to the VM. Note that this would not sacrifice the potential for resource consolidation nearly as significantly as traditional NUMA exposure (see §5.4.1) since a pCPU may sport many SMT contexts between which pCPU resources may be dynamically distributed. In essence, the task of vCPU scheduling is thus largely migrated from the VMM to the hardware itself.

5.4.4 Extended Paravirtualization

Another promising novel approach to the problem of NUMA opacity is the concept of extended paravirtualization, which was recently proposed by Bui et al. [154]. The basis of this technique is traditional NUMA passthrough. Additionally however, a communication mechanism is implemented between the guest and VMM so that the latter can notify the former when it migrates a vCPU between NUMA nodes. This effectively allows for dynamic alteration of the virtual NUMA configuration of the VM at runtime. Whenever this occurs, it is immediately propagated to the scheduler, which may alter its scheduling decisions accordingly. While this technique achieves near-native performance with regard to memory locality, it is a form of paravirtualization, which by definition requires changes to the guest kernel, which in turn constrains its potential for rapid and widespread adoption.

5.5 Related Work

Improving the virtualization process for multithreaded applications has been the subject of active research for many years. This effort has been far from fruitless, since many techniques proposed in literature have gradually evolved into mainstream components of virtualization technologies. To date, there is no shortage of innovative ideas for further improvements which may one day be considered essential components of virtualized systems. Since this chapter largely consists of a reflection on these recently adopted or proposed ideas, any existing work related to mitigating virtualization overhead for multithreaded applications has naturally already been explicitly mentioned above. From this perspective, much of this chapter may be viewed as an extensive reflection on related work.

While most of this chapter is based on known techniques, it adds value by listing them all side by side to provide readers with insight into their individual advantages and drawbacks as well as their relationship to one another. To the best knowledge of the author, no such exhaustive summary of existing and promising future techniques to reduce virtualization overhead for multithreaded applications exists in literature. Additionally, this chapter has presented several novel ideas which are—again to the best knowledge of the author—not described in any existing literature.

5.6 Conclusion

This chapter has discussed many techniques to reduce virtualization overhead for multithreaded applications, specifically within the context of hardware-assisted virtualization of the x86 architecture. Some of these techniques are already widely adopted, but have been shown in this chapter to require further refinement. To the best knowledge of the author, this work is the first to assess these limitations of existing techniques in such depth.

Beyond mainstream technologies, this chapter outlined a wide variety of mitigation techniques proposed in literature. Many of these techniques are still under active development, which makes it safe to say that the virtualization research field still carries plenty of momentum, making a further drastic reduction in the virtualization overhead incurred by multithreaded applications likely in the coming years.

Finally—and perhaps most interestingly—this chapter describes several original ideas of the author, his colleagues and his supervisors. Three of these ideas have been selected for further exploration in this dissertation: paravirtualization of the scheduler tick (§5.1.4), TLB-shootdown aware memory allocator design (§5.3.5) and the adoption of virtualization-friendly application design principles (§5.1.6, §5.2.4 and §5.3.4). Chapters 6, 7 and 8 are respectively dedicated to each of these techniques. These ideas have been chosen for further refinement in this dissertation in favor of some of the other suggestions in this chapter largely because the problem they address and/or the approach they take have received little to no attention from existing literature. Therefore, the author felt that elaborating on these ideas would provide a maximal contribution to the field within the scope of a single Ph. D. project. Moreover, each of the chosen technologies focuses on a distinct level of the system stack, yielding a fully complementary set of improvements. This again maximizes the impact of this dissertation on the state of the art by ensuring that none of the work presented in the later chapters makes previous contributions obsolete.

5.6.1 Personal Contribution

The three ideas chosen for further exploration in this dissertation are original contributions by the author. Other proposed mitigation techniques were either derived from literature or provided by one of the author's supervisors, who is currently actively pursuing some of these.

5.6.2 Future Work

Potential for future work is largely self-evident from the contents of this chapter. For all widely adopted techniques reducing virtualization overhead for multithreaded applications (PLE, halt polling,...), issues warranting further refinement have been revealed. Almost all of the other described techniques require more work before they are ready for widespread deployment. Of all of these techniques, perhaps those with the potential to address user-level spinning most urgently require attention, since §4.2.4 has shown the devastating performance impact of this problem and to date no effective mitigation techniques are available aside from manually replacing user-level spinning primitives with alternate synchronization mechanisms.

Chapter 6

System Amelioration: Paratick

This chapter was previously published as:
S. Schildermans et al. “Paratick: Reducing Timer Overhead in Virtual Machines”. In: *50th International Conference on Parallel Processing*. 2021, pp. 1–10

Timekeeping is a fundamental duty of the OS. This task involves assimilating hardware timekeeping devices and presenting a unified timer API to applications [180]. Additionally, the OS keeps track of the passing of real time in the background and performs general maintenance tasks such as scheduling, accounting, etc. on a regular basis. As described in §4.2.3, contemporary general-purpose OSs drive all of these duties by recurring physical timer interrupts, known as scheduler ticks [181]. The same section details how traditional implementations of this mechanism (referred to hereafter as ’classic periodic ticks’) are often highly inefficient on current (SMP) hardware, while modern implementations thereof (referred to hereafter as ’tickless kernels’) require interaction with the physical timer hardware upon every transition between active and idle CPU states, which may induce excessive overhead in virtualized environments.

§4.2.3 has shown that multithreaded applications making heavy use of blocking synchronization may suffer severely from the virtualization overhead induced by scheduler tick management in tickless systems. Unfortunately, §5.1.4 has made clear that simply reverting to classic periodic ticks in virtualized environments

is not a satisfying solution to this problem, nor is employing any other existing tick management algorithm known to the author. Therefore, an alternative approach to scheduler tick management is highly desirable. §5.1.4 introduced exactly such an alternative approach based on paravirtualization, namely 'virtual scheduler ticks'. This chapter explores this concept as well as the aforementioned problems it aims to resolve in great depth by providing a comprehensive analysis of the shortcomings of existing tick management techniques and detailing, implementing and evaluating virtual scheduler ticks.

Main Findings & Contributions

- This chapter details why neither classic periodic ticks nor tickless kernels perform satisfactorily in virtualized environments;
- The concept of virtual scheduler ticks introduced in §5.1.4 is fleshed out in this chapter;
- This chapter presents and evaluates paratick; an implementation of virtual scheduler ticks in Linux/KVM.

6.1 Background: Timer Management

Many applications (as well as the OS itself) rely heavily on accurate time management. Because programming timer hardware is often complex and expensive, many OSs choose to implement a high level of abstraction in their timer APIs. Most often, application timers are managed as soft interrupts. This means that when an application sets a timer, generally no actual timer hardware is programmed. Instead, the application timer is added to a dedicated data structure (e.g. the 'timer wheel' in Linux [182]). Upon completion of any system call or hardware interrupt, the OS checks if the current system time has surpassed the expiration time of any soft interrupts. If so, it services these interrupts before returning to user space [183]. Therefore, timer management equates to managing the underlying mechanisms that invoke context switches and allow soft interrupts to be serviced. The most important of these mechanisms is the scheduler tick, since the tick ensures that active CPUs are interrupted by a hardware timer (usually the LAPIC timer in x86) at least at the frequency of the tick, which typically lies between 100 and 1000 Hz [156].

As mentioned on several occasions before, the traditional implementation of the scheduler tick involves a timer interrupt on each CPU, recurring at a fixed interval. The handler of this interrupt performs any needed bookkeeping work

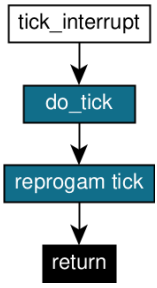


Figure 6.1: Schematic representation of the operation of classic periodic ticks in Linux.

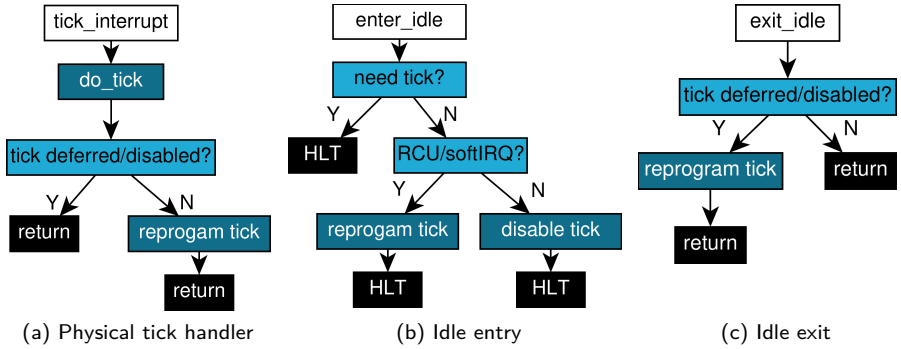


Figure 6.2: Schematic representation of standard tickless kernel operation in Linux.

(handling soft interrupts, scheduling, updating the system time,...) before arming a new tick interrupt and returning. Figure 6.1 displays this process schematically.

While periodic scheduler ticks are simple and effective, they are not suitable for most modern hardware platforms for reasons detailed in §4.2.3. Because of this, Linux 2.6.21 introduced the concept of tickless kernels, later to be adopted by all mainstream OSs [184]. Tickless kernels expand on the concept of classic periodic ticks by identifying scenarios in which the tick is not useful and may consequently be deferred or disabled entirely. Most kernels interpret these 'scenarios in which the tick is not useful' as idle CPUs. Thus, they disable the tick upon idle entry and enable it again upon idle exit. Figure 6.2 describes Linux's implementation of this algorithm. Though details may differ for other OSs, the principle is always similar.

Handling tick interrupts in tickless kernel mode is largely identical to doing so using classic periodic ticks, as shown in figure 6.2a. The only difference between the tickless tick handler and the classic one is that the former checks whether the tick has been deferred or disabled by the time the tick interrupt handler was invoked. This may happen in exceptional circumstances. If so, the reprogramming step is skipped. Figures 6.2b and 6.2c on the other hand represent the core of tickless kernel operation. Whenever a CPU is about to enter the idle loop, the kernel checks if any system component (RCU, irq work,...) explicitly needs the tick to remain enabled or if any RCU events or soft interrupts are due to expire within the next tick period. If so, the tick is not disabled and the CPU immediately enters the idle loop. If not, the algorithm finds the next scheduled RCU callback or soft interrupt. The tick timer is then reprogrammed to expire at the expiry time of that event. If there are none, the tick is disabled entirely. Upon exiting the idle state, the algorithm checks if the tick has been deferred or disabled upon idle entry. If so, it is reprogrammed to expire at the regular tick interval.

As noted in §5.1.4, Linux offers a third option for tick management, namely full dynticks mode. As equally noted in that section however, full dynticks mode may be viewed as a variation on regular tickless operation with the threshold for disabling the scheduler tick on a particular CPU shifted from having no runnable tasks to having one runnable task for that CPU. As such, the findings for tickless kernel operation presented in this chapter are in general equally applicable to full dynticks mode.

6.2 Virtualizing the Scheduler Tick

As alluded to multiple times before, the main issue regarding virtualizing the scheduler tick that it inherently involves hardware interaction. Specifically, Linux uses the TSC for this purpose when possible, since it is the most accurate timer hardware [185]. It is armed by writing the desired expiration time to the `TSC_DEADLINE` MSR, as noted in §4.2.3. When the TSC value reaches said expiration time, the LAPIC generates a local timer interrupt. In native environments, this process has a very low cost. In virtualized environments however, each write to the `TSC_DEADLINE` MSR must be intercepted by the VMM, as its current value may correspond to a timer set by the host or another VM. Moreover, the interrupt generated as the timer expires generates another VM exit, as the VMM must determine the intended recipient. Some VMMs (e.g. KVM) optimize this process by replacing the LAPIC timer by the preemption timer. Namely, upon each VM exit induced by a guest attempting to write to the `TSC_DEADLINE` MSR, the VMM arms the preemption timer for the vCPU in

question, but leaves the `TSC_DEADLINE` MSR untouched. When the preemption timer expires, a special low-cost VM exit is triggered which allows the VMM to inject a timer interrupt [186].

From the above, it is clear that handling scheduler ticks is a costly process in virtualized environments. The magnitude and nature of this cost may however vary greatly depending on the workload and whether the system is employing classic periodic ticks or a tickless kernel. The remainder of this section analyzes virtualization overhead associated with the scheduler tick in a general sense for both of these tick management algorithms.

6.2.1 Classic Periodic Tick

Given that classic periodic ticks have a constant frequency on each vCPU irrespective of its workload, one may intuitively derive that a system hosting a number of VMs n_{VM} employing classic periodic ticks, each having a number of vCPUs n_{vCPU} and a tick frequency f_{tick}, will always incur the following number of VM exits related to timer management over a time period t:

$$VMexits = 2 \times t \times \sum_{n=1}^{n_{VM}} (n_{vCPU} \times f_{tick})$$

The above implies that the host may spend exorbitant resources on processing ticks from guests employing classic periodic ticks when the system is heavily overcommitted. Namely, vCPUs must be suspended whenever a tick arrives for another vCPU, even if the latter is idle [156]. Since one of the main applications of virtualization is consolidation, such OC scenarios where the majority of vCPUs are idle for the majority of the time are not rare. As noted in §5.1.4, this makes classic periodic ticks far from ideal in a virtualized environment.

6.2.2 Tickless Kernels

Tickless kernels are often depicted as almost purely beneficial compared to classic periodic ticks [187, 156]. While in native environments this claim may hold true, in virtualized environments their benefits are less clear. While tickless kernels do reduce the number of timer interrupts generated by lightly utilized VMs, they must reprogram the tick timer upon each idle entry/exit. Since this reprogramming requires a write to the `TSC_DEADLINE` MSR and thus induces a VM exit, the number of VM exits induced by tick management in a tickless system can be described as follows:

$$VMexits = 2 \times t \times \sum_{n=1}^{n_{VM}} \left(L_n \times n_{vCPU} \times f_{tick} + \frac{(1 - L_n) \times n_{vCPU}}{T_{idle}} \right)$$

With L_n the VM load expressed as a ratio of the utilized and maximum VM CPU throughput and T_{idle} the average idle period during the time interval t. Thus, the term $L_n \times n_{vCPU} \times f_{tick}$ represents tick interrupts during active vCPU operation and the term $\frac{(1-L_n) \times n_{vCPU}}{T_{idle}}$ represents the number of transitions between active and idle states during the time interval t.

From the above, it is evident that for tickless kernels to be efficient in virtualized environments, the average idle period T_{idle} must be long relative to the total CPU time spent on idling ($t \times (1 - L_n) \times n_{vCPU}$), which in practice equates to minimizing the number of transitions between idle and active vCPU states since increasing the average idle period ceteris paribus proportionally reduces system throughput, which is obviously not desirable. However, certain types of applications incur many such transitions by design. Examples include multithreaded applications making heavy use of blocking synchronization and I/O-intensive applications. §4.2.3 already discussed the former in detail. Regarding the latter, given that I/O latencies are typically in the range of micro- to milliseconds and most applications block on each I/O transaction [188], I/O performance may suffer significantly in a virtualized environment if the guest employs a tickless kernel. Since the severity of this issue is directly proportional to the frequency of idle transitions and therefore inversely proportional to I/O latency, high-performance I/O devices are affected the most.

6.2.3 To Tick or not to Tick?

The above indicates that both classic periodic ticks and tickless kernels may induce severe performance issues in a virtualized environment. In fact, which of these algorithms is to be preferred depends strongly on the workload W and system settings S_v. To clarify this, let us consider several virtualized systems:

- **S1**: A system hosting a single idle VM with 16 vCPUs;
- **S2**: A system hosting four idle VMs with 16 vCPUs each;
- **S3**: A system hosting a single VM with 16 vCPUs, executing a workload using 16 threads, synchronizing 1000 times per second through blocking synchronization;

- **S4**: A system hosting four VMs with each 16 vCPUs, each executing a workload using 16 threads, synchronizing 1000 times per second through blocking synchronization.

Table 6.1: Number of VM exits induced by classic periodic ticks and tickless kernels in a variety of scenarios.

	S1	S2	S3	S4
periodic ticks	40 000	160 000	40 000	160 000
tickless	0	0	60 000	240 000

Table 6.1 shows the amount of VM exits related to scheduler tick management incurred by each of the above systems when all of the VMs use respectively classic periodic ticks or tickless kernels with a tick frequency of 250 Hz, assuming the workloads are run for 10 seconds on a system with 16 pCPUs. All values are calculated based on the formulas derived in sections 6.2.1 and 6.2.2.

Table 6.1 shows that for low-intensity workloads where the system is mostly idle, tickless kernels are vastly superior to classic periodic ticks. However, for high-intensity workloads which frequently switch between idle and active states, periodic ticks gain the upper hand. Specifically, tickless kernels are preferable as long as the average idle period T_{idle} is longer than the average tick period divided by the number of vCPUs sharing a pCPU. With tick periods commonly ranging between 1 and 10 ms, this is often not the case. Given that parallel computing has become the norm these days and more efficient I/O devices continue to emerge (e.g. datacenter network, NVMe storage,...), demand for better handling of microsecond-level idle periods continues to rise [189]. Moreover, stimulated by workloads such as AI and blockchain, various highly parallel accelerators (e.g. GPGPUs and TPUs) are being designed and deployed. Fine-grained computation offloads to such accelerators incur similarly small idle periods. Thus, neither classic periodic ticks nor tickless kernels meet the requirements of increasingly common highly consolidated virtualized environments hosting I/O-intensive, highly parallel workloads. It is clear that an alternative tick management algorithm is highly desirable.

6.3 Virtual Scheduler Ticks

In an effort to address the issues described above, this dissertation proposes the concept of virtual scheduler ticks, which is a novel tick management algorithm first introduced in §5.1.4. This section details its design and performance implications compared to classic periodic ticks and tickless kernels.

In essence, virtual scheduler ticks views the scheduler tick as a system service managed by the VMM which guests may request through a hypercall interface (see §5.1.4). This effectively equates to paravirtualizing the scheduler tick, which

in turn implies that the guest kernel must be modified so that it no longer programs its own scheduler tick and instead performs the appropriate hypercalls to request ticks from the VMM. The latter may leverage its own scheduler tick interrupts—which many VMMs must program irrespective of any VMs to perform their own bookkeeping work—to inject virtual ticks at the appropriate times. When vCPU execution is resumed, the guest may handle these virtual tick interrupts analogously to how it would process its own physical scheduler ticks. Note however that this relies on the host tick frequency corresponding to (a multiple of) that of the guest, since this is the only way to guarantee vCPUs are interrupted and virtual scheduler ticks are injected at the appropriate time interval. When this is not the case, the host should program the guest preemption timer such that virtual ticks may be injected at the correct rate. Note that this does not introduce meaningful virtualization overhead, since if the guest were to program its own tick interrupts, two VM exits would be generated each tick period for respectively injecting the physical tick interrupt and reprogramming the timer hardware.

The above forms the basic working principle behind virtual scheduler ticks and suffices when the vCPU requiring ticks to be injected is actively running and is therefore regularly interrupted by host scheduler ticks. However, when the vCPU is idle or is sharing the pCPU hosting it with other tasks, the vCPU may be descheduled for long periods of time unbeknownst to the guest and thus not receive any virtual scheduler ticks despite expecting to. Therefore, extra measures must be taken to ensure a virtual tick is delivered to descheduled vCPUs in a timely manner. Concretely, the time of the last virtual tick injection must be accounted for each vCPU. On each VM entry, the host must check if the last virtual tick injection predates the requested tick interval for that vCPU. If so, a virtual tick must be injected and the current time is to be recorded as the last tick. Furthermore, to ensure that idle vCPUs are awoken by the VMM when necessary despite not receiving any virtual scheduler ticks, the guest must check if there are any soft interrupts or RCU tasks scheduled upon idle entry. If so, it must program a timer to expire at the expiration time of the earliest of these events. We decide not to disable this timer upon exiting the idle state, as the overhead induced by a single timer is negligible and it is likely that the vCPU will re-enter an idle state before the timer has expired. If the timer were to be disabled upon idle exit, it would likely need to be reprogrammed upon idle entry, thus inducing two unnecessary VM exits.

While the concept of virtual scheduler ticks as proposed above may still induce some VM exits, this number is negligible compared to both classic periodic ticks and tickless kernels for almost any workload. Concerning the former, in particular when guests are mostly idle and/or the host is overcommitted this may lead to a tangible performance improvement. Concerning the latter,

virtual scheduler ticks is guaranteed to reduce the number of VM exits upon idle entry and exit, as tickless kernels require the timer hardware to be touched on practically every transition between active and idle states. When vCPUs are actively running on the other hand, even in the worst-case scenario where the host tick frequency is vastly lower than that of the guest and consequently (almost) all virtual scheduler ticks must be triggered via the preemption timer, virtual scheduler ticks reduces the amount of required VM exits by half because one VM exit suffices to inject and process a virtual scheduler tick, while §6.2 has made clear that for physical ticks, the same operations require two VM exits. Notwithstanding, the benefits of virtual scheduler ticks compared to tickless kernels mostly depend on the workload. Within the context of this dissertation, being multithreaded workloads, system throughput may improve drastically for applications relying heavily on blocking synchronization, as noted in §4.2.3. Nevertheless, application execution times may not improve accordingly because it is determined solely by the critical path [151]. Therefore, only VM exits incurred upon idle exit (idle entry is by definition not part of the critical path) and belonging to a single execution path influence application execution time. Thus, for multithreaded workloads, a significant improvement in $\delta\eta_r$ is expected, which may however translate to a much smaller improvement in $\delta\eta_t$ as ω is likely low for this particular form of virtualization overhead. Additionally however, §6.2.3 identified (sequential) I/O-intensive workloads as likely benefiting from improved scheduler tick management. For such workloads, virtual scheduler ticks may indeed improve both $\delta\eta_r$ and $\delta\eta_t$ significantly since ω is likely to be much higher in comparison. Namely, for these applications almost all VM exits incurred upon idle exit—and if I/O latencies are sufficiently low even those upon idle entry—are likely part of the critical path as any delay in processing an I/O interrupt likely delays the next I/O operation.

6.4 Paratick

Because this dissertation aims to reach beyond purely theoretical reasoning and seeks to provide tangible improvements to the state of the art based on (a subset of) the novel techniques to mitigate virtualization overhead for multithreaded applications it proposes, we developed a prototype implementation of virtual scheduler ticks based on Linux/KVM (kernel 5.10.26) under the name 'paratick'. Paratick is freely available[1] and documented in abstract terms below. Refer to appendix A for a complete transcript of its source code.

[1] `https://github.com/StijnSchildermans/paratick.git`

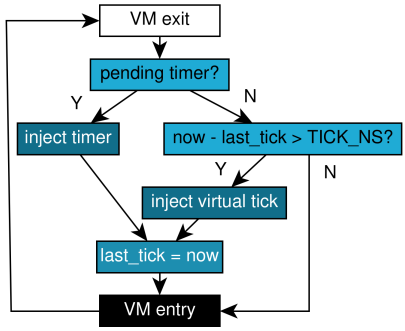
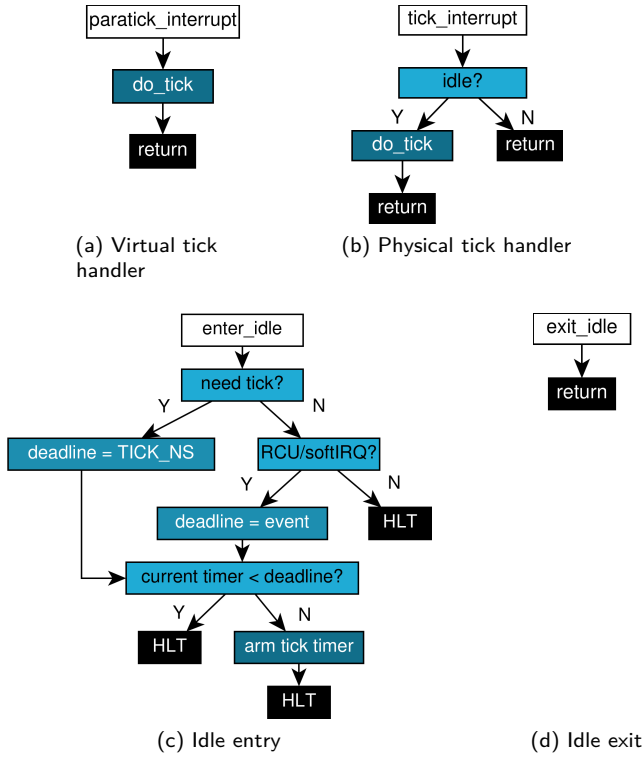


Figure 6.3: Schematic overview of host-side paratick code.

6.4.1 Host

Implementing paratick requires minimal effort on the host side. Firstly, a field named `last_tick` was added to the struct KVM uses to represent a vCPU internally (`kvm_vcpu`), recording the time of the last virtual tick injection. Secondly, the main KVM loop which is responsible for executing vCPUs was modified. If the vCPU has a pending local timer interrupt upon VM entry, the `last_tick` field of the `kvm_vcpu` struct is updated. Paratick thus heuristically assumes that the local timer interrupt to be injected was programmed by the guest-side paratick code upon idle entry. This assumption is acceptable since Linux by default performs basic timekeeping work upon receipt of any interrupt, even when the interrupt itself has nothing to do with the scheduler tick [115]. Moreover, extensive testing has not revealed any negative side effects of this optimization. If no local timer interrupt is pending upon VM entry on the other hand, paratick evaluates if the time elapsed since the last tick injection is greater than the tick period. If so, a virtual tick interrupt is injected and the `last_tick` field of the `kvm_vcpu` struct is updated. Paratick uses interrupt vector `235` for this purpose. Figure 6.3 illustrates all of this schematically.

To demonstrate the potential of virtual scheduler ticks, the above host-side modifications suffice since the host and guest are guaranteed to have the same tick frequency. However, when this can not be guaranteed, a hypercall must be implemented so that the VM can request virtual scheduler ticks at a different frequency. To deliver these ticks, the VM entry code must be modified further to program the preemption timer upon each VM entry to guarantee timely virtual tick delivery. These features were omitted in paratick because they do not add value from a research perspective. Note that any implementation of virtual scheduler ticks aiming for widespread adoption should however contain them.



(a) Virtual tick handler

(b) Physical tick handler

(c) Idle entry

(d) Idle exit

Figure 6.4: Schematic representation of guest-side paratick code.

6.4.2 Guest

The guest-side implementation of paratick is somewhat more pervasive than its host-side counterpart. Still, altering just the main scheduler tick source file (`kernel/time/tick-sched.c`) suffices. Figure 6.4 schematically shows the high-level guest-side paratick implementation, arranged in such a way that it can easily be compared to the regular tickless Linux kernel, as shown in figure 6.2.

Figure 6.4 shows that paratick preserves the basic structure of the tickless Linux kernel, while adding an extra handler for virtual tick interrupts. Below, all guest-side implementation details of paratick are described step by step.

System Boot

Both the regular tickless kernel and paratick are built on top of the standard Linux `hrtimer` API [115]. Unfortunately however, this API is initialized relatively late in the boot process. Before this time, the system must use a traditional periodic scheduler tick. Therefore, the paratick initialization code is integrated with the standard tickless initialization code and any virtual scheduler ticks arriving before this code has been executed are rejected. The initialization code itself encompasses installing an interrupt descriptor for the virtual scheduler tick interrupt vector and disabling the aforementioned temporary periodic scheduler tick.

Virtual Tick Handling

As figure 6.4a shows, paratick employs a dedicated handler for virtual scheduler ticks, which slightly differs from the tick interrupt handler employed by the tickless Linux kernel shown in figure 6.2a. Namely, under no circumstances does it rearm the tick timer, since this responsibility has been delegated to the VMM.

Physical Tick Handling

As described in §6.3, paratick may require a physical timer to be programmed upon idle entry. Figure 6.4b shows the handler for this physical timer interrupt. It first checks if the vCPU is still idle when receiving the interrupt. If so, this interrupt is likely crucial to the system and is treated as a virtual tick interrupt. If not, the vCPU is currently operating normally, meaning virtual scheduler ticks are actively being injected. There is thus no need to perform any work and the handler returns.

Idle Entry

The main challenge in implementing paratick has proven to be determining whether a physical timer should be set upon idle entry. Thankfully, paratick can largely recycle tickless kernel idle entry code for this purpose, as is evident by comparing figures 6.4c and 6.2b. Note however that the status quo for paratick is that no timer is programmed and the idle entry code must check whether a timer should be set, while the status quo for tickless operation is that a timer is set and the idle entry code should determine whether to disable it. Thus, if the tickless code determines the tick must be retained, paratick programs a timer

to expire at the regular tick interval. Otherwise, it checks if a timer must be set at the expiry time of the next RCU event or soft interrupt, again recycling existing tickless kernel code. If so, the determined deadline is compared to the current expiry time of the physical tick timer, since as described in §6.3, the timer may not yet have expired after having been set at a previous idle entry. Only if the physical tick timer is not running or the newly determined expiry time is sooner than its current one, it is (re)programmed.

Idle Exit

Because as described in §6.3 we heuristically determined that it is beneficial not to disable any physical timers set at idle entry upon idle exit, no action must be taken when a vCPU returns from idle, as shown by figure 6.4d. This stands in contrast to the tickless kernel implementation in Linux, which must re-enable the tick timer at (almost) each idle exit (see figure 6.2c).

6.5 Evaluation

Having developed a prototype implementation of virtual scheduler ticks, it is possible to provide concrete evidence of its performance benefits by empirically comparing it to the state of the art. To this end, experiments were set up in accordance with the prescriptions provided in §3.3. The baseline OS for both the host and the guest is Ubuntu 20.04, employing Linux 5.10.26 in in the default tickless configuration. Since kernels using classic periodic ticks are rare these days and classic periodic ticks were already compared to tickless kernels in §6.2.3, this section omits directly comparing paratick to classic periodic ticks. Readers may nevertheless infer such a comparison from combining the results in this section with those presented in §6.2.3. This decision also simplifies the evaluation process, as §6.3 has made clear that the benefits of virtual scheduler ticks over classic periodic ticks only clearly manifest themselves in OC environments, while its benefits over tickless kernels are equally profound in UC settings. As such, limiting the evaluation to UC environments suffices here. Furthermore, §6.3 identifies the main workloads of interest for this evaluation: multithreaded and I/O-intensive applications. However, because a fair performance assessment must include at least some workloads for which virtual scheduler ticks is not expected to provide a meaningful performance improvement, sequential, computation-intensive applications are included as well. Thus, this section evaluates the performance of paratick compared to a state-of-the-art tickless kernel in an UC setting for sequential, multithreaded and I/O-intensive applications.

Because the intent of this section is to demonstrate the potential performance benefits of virtual scheduler ticks as accurately as possible, rather than faithfully assessing virtualization overhead in se (as was the case in chapter 4), it is prudent to alter some system settings that may distort experimental results; in particular PLE and halt polling. Namely, the former is only beneficial in OC environments (see §3.2.6). In UC scenarios, any VM exits triggered by PLE unnecessarily degrade performance. Regarding the latter, §5.1.1 has shown that halt polling may drastically increase $\delta\eta_r$ in an effort to slightly improve $\delta\eta_t$. This may obfuscate the benefits of virtual scheduler ticks since in some cases, a more efficient execution may lead to seemingly worse performance when using halt polling, as it may increase thread contention, which leads to increased polling cycles without improving execution time tangibly. Therefore, both PLE and halt polling were disabled for all experiments documented in this section.

Following the reasoning laid out in the previous paragraph, it is important to note that the results in this section do not accurately reflect virtualization overhead, but rather potential performance improvements associated with virtual scheduler ticks. As such, all results are presented as the result paratick yields for a given metric relative to the result yielded by its tickless counterpart. To clearly make this distinction, we choose not to represent any results in terms of $\delta\eta_r$ or $\delta\eta_t$. Instead, this section employs the following metrics:

- **VM exits**: Since paratick aims to eliminate the majority of writes to the `TSC_DEADLINE` MSR and associated VM exits, assessing the number of VM exits shows to what extent paratick achieves its basic goal. This metric may be measured directly using `perf`;
- **System throughput**: System throughput shows the effect of paratick on system resources. This metric may be viewed as a proxy for $\delta\eta_r$, with the important distinction that it more clearly shows the total amount of resources consumed—by useful work and overhead alike—which more easily allows for placing the performance improvements paratick yields into perspective. Rather coincidently, within the context of this chapter, throughput may be measured analogously to $\delta\eta_r$ in the context of multithreaded applications, i.e. in terms of CPU cycles (see §3.1.1). Namely, although throughput is determined by many factors, §6.3 makes it clear that virtual scheduler ticks aims to improve system performance solely through eliminating certain VM exits, which frees up CPU resources for other tasks. Therefore, the reduction in CPU cycles paratick achieves represents the maximum throughput improvement it may yield;
- **Execution time**: Analogously to system throughput, execution time serves as a proxy metric for $\delta\eta_t$ which indicates paratick's performance benefits visible to end users. This metric is directly measurable.

Table 6.2: Average performance improvement of paratick accross all PARSEC benchmarks in sequential mode.

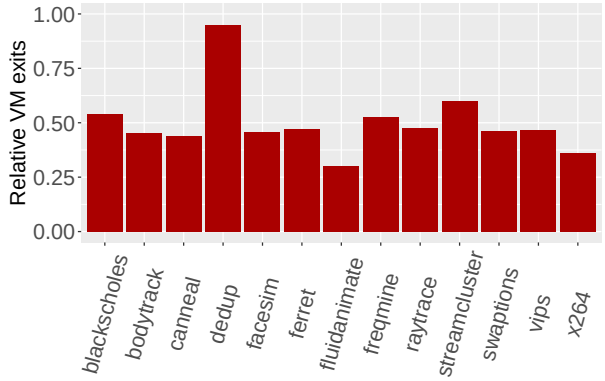
VM exits	System throughput	Execution time
-50%	+7%	-2%

6.5.1 Sequential Workloads

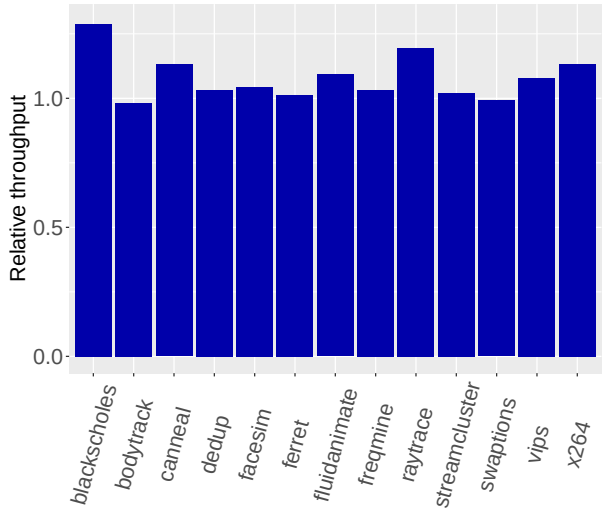
As described above, paratick is not expected to benefit computation-intensive sequential workloads. Conversely, any overhead introduced by paratick itself would likely still be measurable because these workloads obviously still require scheduler ticks to be injected. Because of this, assessing these workloads allows for estimation of the gross cost of paratick, irrespective of potential performance gains. Concretely, figure 6.5 shows the performance of paratick relative to a standard tickless Linux kernel for each of the PARSEC workloads run in sequential mode on a VM with a single vCPU. To facilitate interpretation of this figure, table 6.2 shows the aggregated results for all PARSEC benchmarks.

Figure 6.5a shows that even for low-intensity workloads, paratick reduces the number of VM exits drastically compared to a plain tickless kernel. This is to be expected, since such workloads induce very few VM exits to begin with, as may be derived from §6.2. A large portion of these few VM exits are caused by three operations: arming the guest tick timer, delivering host ticks and delivering guest ticks. Since paratick eliminates 2 of these 3 major causes of VM exits, it greatly reduces virtualization overhead for low-intensity workloads.

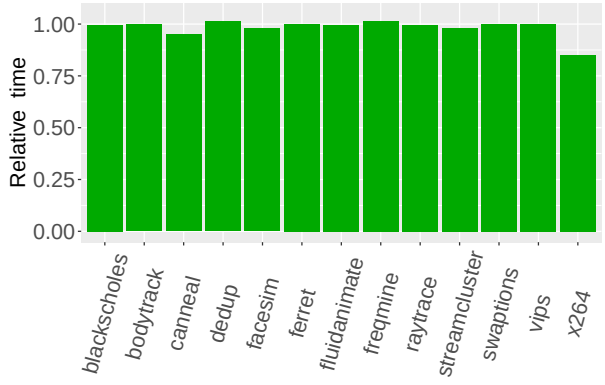
Despite figure 6.5a showing excellent results, figures 6.5b and 6.5c indicate that paratick only marginally improves system throughput and application performance for low-intensity workloads. This is however in line with the expectations laid out in §6.3: even though the number of VM exits is reduced drastically, the amount of resources spent processing them is negligible relative to those spent on the workload itself. More importantly, these figures show that even in scenarios where paratick offers negligible benefits, workload latency and system throughput are not affected negatively, indicating that the gross cost of paratick is minimal.



(a) VM exits



(b) System throughput



(c) Execution time

Figure 6.5: Relative performance of paratick compared to tickless Linux for sequential PARSEC workloads.

Table 6.3: Average performance improvement of paratick accross all PARSEC benchmarks in all tested scenarios.

VM size	VM exits	System throughput	Execution time
Small	-42%	+12%	-1%
Medium	-47%	+13%	-3%
Large	-44%	+16%	-1%

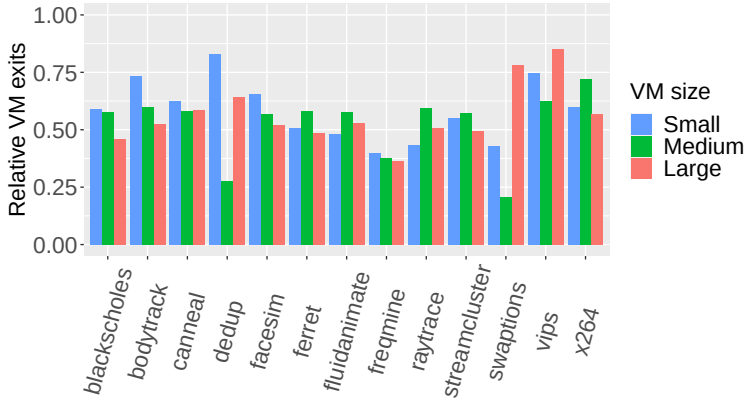
6.5.2 Multithreaded Workloads

Having established that paratick does not introduce tangible gross overhead, the magnitude of its potential benefits may by assessed using workloads outlined in §6.3 as conceptually greatly profiting from virtual scheduler ticks, the first category of which being computation-intensive multithreaded applications. Three distinct system settings S are evaluated:

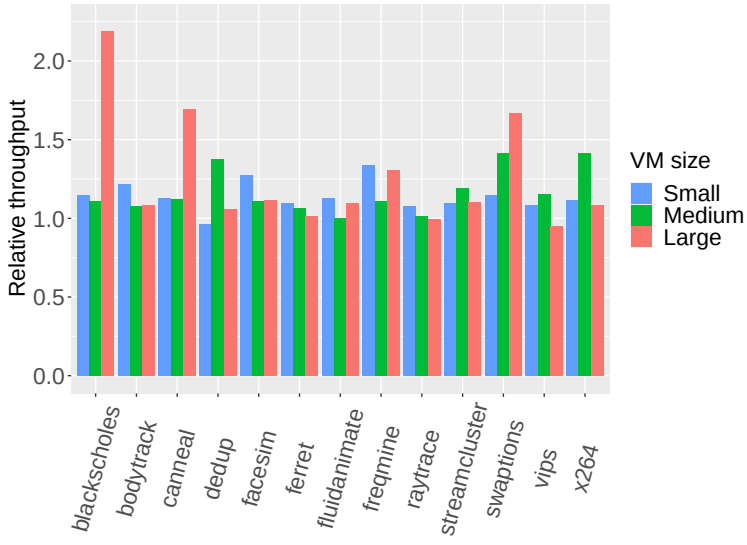
- A **small** VM with 4 vCPUs collocated on the same NUMA node;
- A **medium** VM with 16 vCPUs spread over 2 NUMA nodes;
- A **large** VM with 64 vCPUs spread over 4 NUMA nodes.

In each of the above scenarios, the PARSEC benchmark suite was evaluated with the level of parallelism set equal to the number of vCPUs sported by the described VM. All metrics are measured as in §6.5.1. Equally analogously to §6.5.1, figure 6.6 displays the results for all individual benchmarks and table 6.3 shows the aggregate results across all of the benchmarks in each scenario.

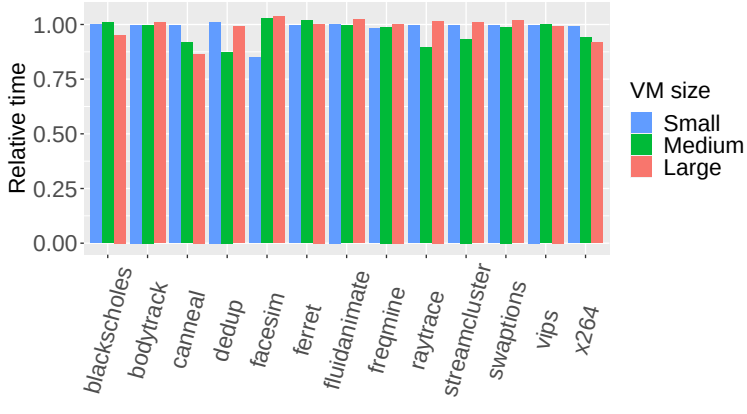
Figure 6.6a shows that for multithreaded workloads, paratick reduces the relative number of VM exits compared to tickless kernel operation by roughly the same amount as for sequential ones. Nevertheless, figure 6.6b indicates that for several of these workloads—in contrast to sequential ones—this translates to a drastic improvement in system throughput. This is not illogical, since chapter 4 has demonstrated that multithreaded workloads induce many more VM exits than their sequential counterparts. This means that the same relative reduction in VM exits translates to a comparatively much greater performance improvement. However, this improvement varies greatly between benchmarks and system configurations. This is to be expected, since as outlined in §6.3, virtual scheduler ticks specifically reduces the cost of blocking synchronization. Not all multithreaded workloads rely on this mechanism to the same extent. Furthermore, the effectiveness of paratick tends to increase with vCPU count because as the level of parallelism increases, so do thread contention and consequently switches between running and idle vCPU states.



(a) VM exits



(b) System throughput



(c) Execution time

Figure 6.6: Relative performance of paratick compared to tickless Linux for multithreaded PARSEC workloads.

Table 6.4: Average performance improvement of paratick across all tested phoronix-fio benchmarks.

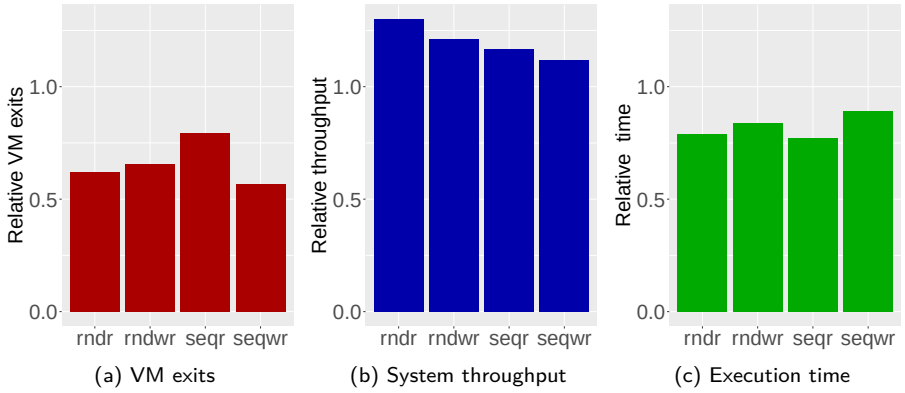
VM exits	System throughput	Execution time
-34%	+20%	-18%

On a somewhat less positive note, figure 6.6c confirms that as anticipated in §6.3, the large throughput gain shown in figure 6.6b does not translate to a comparable reduction in application execution times, implying that the VM exits eliminated by paratick are mostly not part of the critical path for multithreaded workloads. Nevertheless, improved throughput in itself is highly beneficial since in scenarios where system resources are saturated, resource availability dictates the execution time of the critical path and thus of the entire application. Moreover, considering throughput is measured in terms of CPU cycles in this section, increased throughput implies increased efficiency and thus reduced energy consumption.

6.5.3 I/O-Intensive Workloads

Besides multithreaded workloads, §6.3 describes I/O-intensive applications as potentially greatly benefiting from virtual scheduler ticks. This section assesses the veracity of this claim using a dedicated I/O benchmark, namely the *fio* benchmark from the Phoronix benchmark suite [125]. This benchmark was executed on a VM with one vCPU, configured analogously to the VM employed in §6.5.1. Sequential read (seqr), sequential write (seqwr), random read (rndr) and random write (rndwr) performance were independently evaluated. For each of these tests, block sizes were varied between 4 kB and 256 kB. The `sync` I/O driver was used, as synchronous I/O is much more popular than its asynchronous counterpart due to the complexity of the latter [188]. Direct I/O was disabled as is common practice. Buffering I/O was disabled as well to simulate reading/writing large data sets. Again analogously to §6.5.1 and §6.5.2, figure 6.7 shows the results for each category individually while table 6.4 shows the aggregated results of all categories.

Figure 6.7a indicates that also for I/O-intensive workloads, paratick significantly reduces VM exits. This reduction is however somewhat smaller than for the application classes discussed above. This is to be expected, because I/O is notorious for inducing high virtualization overhead in general [66] and the test system does not possess a high-end SSD device supporting SR-IOV. Therefore, timer-related VM exits make up a relatively small subset of the total number of VM exits such workloads induce. However, figure



(a) VM exits (b) System throughput (c) Execution time

Figure 6.7: Relative performance of paratick compared to tickless Linux for I/O-intensive workloads.

6.7b indicates that this comparatively small reduction in VM exits yields a significant throughput improvement for I/O-intensive applications. Interestingly, the average throughput improvement displayed in table 6.4 is not much lower than the average reduction in VM exits. This confirms that processing VM exits consumes a significant fraction of the total system resources utilized by I/O-intensive applications. Even more impressively, figure 6.7c and table 6.4 reveal that for I/O-intensive applications, throughput improvement translates almost directly to improved application execution times. This makes sense, since as described in §6.3, at least half of the VM exits eliminated by paratick are part of the critical path for these workloads. Note that figure 6.7c indicates that read operations benefit the most from paratick. Given that read latencies are lower than write latencies and reads are mostly synchronous while writes are generally asynchronous, reads induce more frequent switches between active and idle vCPU states than writes. Therefore, the VM exits eliminated by paratick make up a larger percentage of the total application execution time for read-heavy workloads and by extension I/O operations with low latencies in general, which are likely to become ever more prevalent towards the future.

6.6 Related Work

Timer overhead in virtualized environments has received little attention in literature. Only a few papers [190, 191, 192] target timekeeping in VMs and its effects on scheduling and application performance [193, 194, 195]. One major reason for this is that most recent efforts regarding reducing virtualization overhead focused on more dominant forms thereof [10], including LHP, BWW, LWP, TLB shootdown preemption, etc. However, as stated before, recent improvements to virtualization technology have largely mitigated these issues. This makes optimizing timer management one of the last significant remaining challenges regarding efficient virtualization of the x86 platform.

Although the problem of scheduler tick management in virtualized environments has to the best knowledge of the author never been addressed explicitly in literature, some studies indirectly offer potential solutions. OSv [85], a novel unikernel-based OS designed specifically for cloud computing employs a fully tickless design, utilizing a high resolution clock for time accounting as long as the use case only calls for a single application to be run at a time. While OSv is able to outperform a traditional Linux system by up to 47% in some aspects and therefore far exceeds the performance gains paratick achieves, it is not a general-purpose OS and achieves these gains by sacrificing many traditional OS capabilities. While for many cloud applications such a design suffices, it is obviously not a generalizable solution.

A more widely applicable solution to excessive timer overhead is 'direct interrupt delivery (DID)' [163]. DID directly delivers timer interrupts to the target VM, bypassing VM exits through clearing the 'external interrupt exiting (EIE)' flag in VMCS. In addition, it programs the hardware not to perform VM exits upon writes to the `TSC_DEADLINE` MSR. While the authors of [163] claim a VM throughput improvement of up to 67%, timers set by the VMM and descheduled vCPUs are restricted to a designated CPU, which can become a bottleneck under heavy loads. Moreover, the designated core can not be used by VMs, which can be interpreted as a static virtualization overhead inversely proportional to the number of pCPUs in the system. Additionally, DID only achieves 67% throughput improvement for one particular workload (memcached). For other I/O-intensive workloads, [163] reports much more modest improvements of around 10%. Taking into account the throughput loss due to the aforementioned dedicated CPU for timer management and the fact that many workloads (e.g. sequential and memory-intensive tasks) do not benefit noticeably from improved timer performance to begin with, it is clear that DID is a specialist tool drastically benefiting specific workloads while negatively affecting others. Paratick in contrast is generally applicable since it has no (known) negative effects on the system.

6.7 Conclusion

Even in state-of-the-art virtualized environments, timer management remains a major source of virtualization overhead. This chapter elaborated on the concept of virtual scheduler ticks, which was first introduced in §5.1.4 as a technique to address this issue through the use of paravirtualization. Moreover, this chapter has shown the potential of this concept by detailing a prototype implementation thereof in Linux/KVM and demonstrating that it may greatly enhance system throughput by eliminating most VM exits related to scheduler tick management. Especially multithreaded applications relying heavily on blocking synchronization and I/O-intensive applications benefit. For the former, this system throughput improvement translates to only a minor application execution time reduction, since many of the VM exits eliminated by virtual scheduler ticks are not part of the critical path of these workloads. For the latter however, performance gains are in accordance with system throughput amelioration.

To the knowledge of the author, virtual scheduler ticks is the only generally applicable solution to the problem of excessive timer-related virtualization overhead. The only major drawback of virtual scheduler ticks is its reliance on paravirtualization and associated requirement for modifications to the guest kernel. This complicates dissemination, especially towards closed-source systems. Whenever this drawback is not a concern however, virtual scheduler ticks is a clear improvement over tickless kernels and classic periodic ticks alike in virtualized environments.

6.7.1 Personal Contribution

All of the work presented in this chapter was performed by the author of this dissertation. Nevertheless, his supervisors provided him with valuable feedback throughout the course of the research this chapter documents.

6.7.2 Future Work

The obvious avenue for future work based on virtual scheduler ticks is developing a more refined version of paratick, from which a patch for the mainline Linux kernel may be proposed. Specifically, paratick does not yet incorporate the hypercall interface proposed in §6.3 which would allow it to support guests with arbitrary tick frequencies. Moreover, more testing (and likely refinement) is needed to ensure paratick is stable under all circumstances.

Chapter 7

Runtime Amelioration: PTLBMalloc2

This chapter was previously published as:
S. Schildermans et al. “Ptlbmalloc2: Reducing TLB Shootdowns with High Memory Efficiency”. In: *ISPA-BDCloud-SocialCom-SustainCom 2020* (2020), pp. 76–83

A fundamental fact about computer science that appears to be overlooked for too often in virtualization research is that even a perfectly designed system is only as efficient as the applications it executes. Chapter 5 has regularly alluded to this by suggesting application-level solutions to many of the remaining challenges regarding virtualizing multithreaded applications. Particularly interesting in this regard is the issue of TLB consistency (see §3.2.7), because as §5.3 lays out, many solutions to this problem have been proposed at hardware and system level but none have attained widespread adoption to date. Simultaneously, the same section explains that while this problem may perfectly be tackled at application level, this approach seems to have been neglected so far in literature. This chapter aims to rectify this oversight.

When considering the relationship between applications and TLB shootdown overhead, memory allocators spontaneously come to mind. After all, these runtime components determine to a large extent how the application interacts with the virtual memory subsystem (and by extension the TLB), often entirely transparently higher-level application code. This implies that memory allocators

are to a large extent in control of the number of TLB shootdowns an application induces and ameliorating their behavior with regard to TLB shootdowns is likely to significantly improve the performance of multithreaded applications in a virtualized context.

As laid out in §5.3.5, contemporary memory allocators do not consider minimizing TLB shootdown overhead as a principal design goal. While this used to be acceptable since TLB shootdowns are satisfactorily efficient in simplistic legacy systems, §4.2.3 has made clear that this does no longer hold true in modern highly consolidated virtualized many-core NUMA environments. As such, §5.3.5 suggests that a memory allocation paradigm incorporating TLB consistency as a fundamental design trade-off rather than a side note could be the key to addressing excessive TLB shootdown overhead on modern systems without significantly affecting other performance metrics. This chapter is dedicated to devising exactly such a paradigm. Additionally, it provides a prototype implementation thereof based on ptmalloc2 and presents evidence for its performance benefits over traditional memory allocators through controlled experiments. Before all of this however, it dives deep into the performance implications of TLB shootdowns and how existing memory allocation paradigms (fail to) address them.

Main Findings & Contributions

- This chapter quantifies TLB shootdown overhead with respect to several system properties and shows that this is a growing issue;
- This chapter identifies the 'arena imbalance issue', which may cause excessive TLB shootdowns in contemporary efficiency-focused memory allocators;
- This chapter details the concept of global hysteresis, which has been first introduced in §5.3.5;
- This chapter presents and evaluates ptlbmalloc2: an implementation of global hysteresis built as a C library on top of ptmalloc2.

7.1 Background: TLB Shootdown Causes

Previous chapters have already described the internal mechanics of TLB shootdowns and how they may degrade system performance. However, equally crucial to addressing TLB shootdown-induced performance degradation is understanding which mechanisms trigger these shootdowns in the first place. In a general sense, these include any operation that alters one or more PTE(s). Such operations may originate from the system itself on the one hand or from an application request in the form of system calls on the other. Regarding the former, the following dominate [196, 197]:

- **Transparent huge pages**: Historically, the size of memory pages was almost always 4 kB. In recent years however, applications often require so much memory that a 4 kB page size has become impractical for several reasons, most importantly a high TLB miss rate. By increasing the page size, a single PTE covers a larger address range, which may greatly reduce TLB miss rate and associated page walk overhead. Therefore, modern Linux kernels use a page size 2 MB rather than 4 kB whenever possible, entirely transparently to applications. This process is called 'transparent huge pages' [198]. One method the kernel employs to achieve this is scanning the memory space looking for sets of 512 contiguous 4 kB pages belonging to the same virtual address space. If it finds such a set, it promotes these pages to a single 2 MB page and purges references to the old 4 kB pages from the TLBs;
- **Page migrations**: As mentioned earlier in this dissertation, optimizing memory locality is critical on NUMA systems. This means that on such systems, the kernel may dynamically migrate memory pages from one NUMA node to another when it deems doing so beneficial. This evidently changes the physical addresses of the migrated pages, enforcing a TLB shootdown;
- **Memory compaction**: When memory fragmentation becomes problematic, the kernel may move allocated pages directly adjacent to one another, merging any free space between them. The relocated pages must be purged from the TLBs by means of a TLB shootdown;
- **Memory deduplication**: In virtualized systems, certain areas of the memory space are likely identical across guests. Examples include kernel code and shared libraries. Some VMMs merge these pages, which improves memory efficiency. During this merging process, references to the duplicates of a page must be purged from the TLBs;

- **Memory reclamation**: When the system is low on memory, the kernel may free parts thereof without application consent. Most often, the freed pages are written to disk and their PTEs are removed from the page table and TLB alike. When an application attempts to access a reclaimed page, a page fault occurs and the kernel restores it;
- **Page cache write-back**: Linux buffers reads from disk in memory for performance reasons. When a buffered page is written to, it is marked as dirty and the change is propagated to disk asynchronously. This dirty mark must be propagated to all CPUs that may hold a reference to the PTE in question by means of a TLB shootdown;
- **Copy-on-write**: When an application writes to a copy-on-write page, the kernel immediately alters the PTE pointing to that page so that it points to the new copy thereof and updates the TLBs accordingly.

From the above, it is clear that TLB shootdowns are essential to many system processes. Notwithstanding, these system-induced TLB shootdowns rarely cause noteworthy performance degradation. Applications on the other hand may induce an arbitrary number of TLB shootdowns by performing any system call that reduces their access to memory in any way [115]:

- `(s)brk`: `Brk` and `sbrk` are both used to change the location of the program break (the former sets it at an address provided by the caller while the latter increments it by the amount provided by the caller), which is a rudimentary yet efficient way of altering the memory space available to the application. In particular when the memory space shrinks, a TLB shootdown is required to avoid illegal memory accesses by other CPUs;
- `munmap`: This system call returns an address range to the system. Said address range is invalidated and the page table and the TLBs are updated;
- `madvise`: This system call gives the kernel advice about certain properties of an address range. Although `madvise` is used for many purposes (see [199]), regarding TLB shootdowns its most important use is `madvise(MADV_DONTNEED)`, which informs the kernel that the memory range passed by the caller may be freed whenever the kernel sees fit. In contrast to `munmap`, the address range remains valid but the physical pages backing it are discarded. The PTEs associated with the page range are consequently removed immediately, inducing a TLB shootdown;
- `mprotect`: This system call changes the access permissions of a memory range. These permissions are stored in the page table, which means that relevant PTEs and TLBs require updating.

It does not require much insight to realize that the above system calls are all crucial to virtual memory management and are therefore essential ingredients for any application memory allocator. This immediately clarifies the impact efficient memory management may have on TLB shootdowns, irrespective of the many system-level causes of the latter.

7.2 TLB Shootdown Cost

While chapter 4 already provided some insight into the virtualization overhead induced by TLB shootdowns, many aspects of their performance implications remain unclear. After all, virtualization is far from the only factor that may influence the cost of TLB shootdowns. Having obtained a deep understanding of TLB shootdowns during previous chapters, two such factors intuitively emerge:

- **CPU count**: Since TLBs are local to each core and a TLB shootdown must flush all TLBs potentially containing the target entry, the number of IPIs required to perform a TLB shootdown linearly increases with CPU count;
- **NUMA architecture**: IPI latency vastly increases when the target CPU is located on another NUMA node because the interrupt signal must travel through the motherboard rather than just the CPU bus. Since as made clear in §3.2.7 the CPU performing a TLB shootdown must wait until all recipient CPUs have acknowledged IPI receipt, if only one of the recipients is located on a remote NUMA node, performance may deteriorate significantly.

This section aims to complement the knowledge gathered earlier in this dissertation regarding the performance implications of TLB shootdowns through analyzing the impact of the above system properties on their cost. To this end, several experiments were performed based on the guidelines provided in §3.3. A custom microbenchmark specifically designed to induce as many TLB shootdowns as possible was chosen as the workload for all of these experiments, so as to minimize interference of operations which are currently not of interest. Listing 7.1 shows the source code of this microbenchmark.

```
void* madv(void* mem){
  for (long i=0;i<1000000;i++)
    madvise(*((char**) mem),4096,MADV_DONTNEED);
}
```

```
int main(int argc, char** argv){
  void *mem;
  posix_memalign(&mem, 4096, 8192);
  pthread_t threads[16];

  for (int i = 0; i<16;i++)
    pthread_create(&threads[i], NULL, madv, &mem);
  for (int i = 0; i<16;i++)
    pthread_join(threads[i], NULL);

  return 0;
}
```

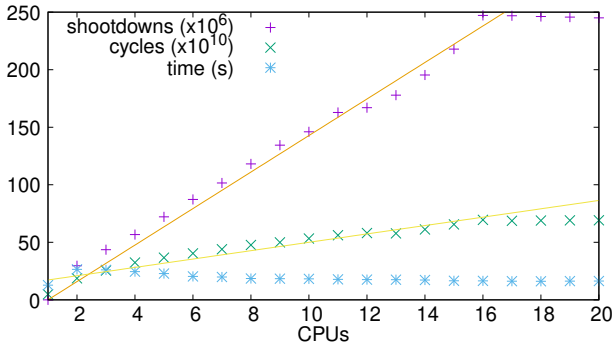
Listing 7.1: Microbenchmark generating many TLB shootdowns.

Listing 7.1 creates 16 threads, all performing the TLB shootdown-inducing system call `madvise(MADV_DONTNEED)` in a loop. Its performance in function of the system properties identified above is measured primarily in the form of CPU cycles and application execution time, in keeping with previous chapters.

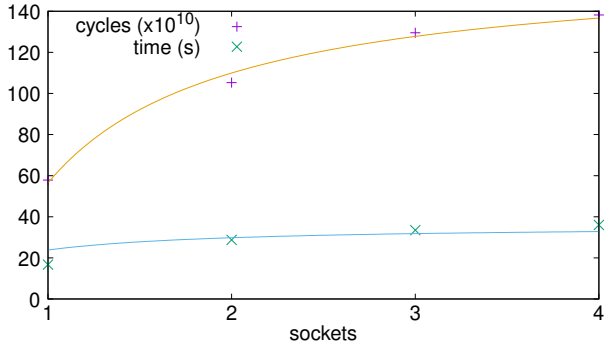
7.2.1 CPU Count

The impact of CPU count on TLB shootdown overhead may be analyzed by pinning the microbenchmark from listing 7.1 to a single socket with CPU counts varying between 1 and 20. Figure 7.1a shows the results of this experiment.

Figure 7.1a indicates that when the benchmark is run on a single CPU, no TLB shootdown IPIs are sent, as intuitively expected. For higher CPU counts, the number of IPIs sent increases linearly. This illustrates the limited capacity of the system to optimize the number of IPIs required for a TLB shootdown. Namely, the OS must send IPIs to all cores sharing the virtual address space of the initiating CPU to guarantee correctness, regardless of whether those cores actually contain the entry to be purged. Above 16 cores however, the number of TLB shootdown IPIs stabilizes, as the benchmark can not use more than 16 CPUs simultaneously. The number of cycles in figure 7.1a reflects the increase in IPIs, indicating a linear relationship between the number of CPUs concurrently used by a program and the system effects of the TLB shootdowns it generates. At first glance, execution time does not seem to follow this trend. Note however that the total amount of work the benchmark performs remains constant for all CPU counts. Ideally, one would thus expect the execution time of the benchmark to be inversely proportional to the CPUs it utilizes, which is clearly not the case in figure 7.1a. Thus, both the system and application effects of TLB shootdowns drastically increase with CPU count.



(a) CPU count



(b) Socket count

Figure 7.1: Impact of several system properties on TLB shootdown overhead.

7.2.2 NUMA

To assess the impact of NUMA on TLB shootdown cost, listing 7.1 was run pinned to 12 cores, spread over 1 to 4 sockets. Figure 7.1b shows the results.

According to figure 7.1b, indeed both execution time and CPU cycles rise with the number of sockets. As noted before, this is a consequence of IPIs sent to a remote NUMA node exhibiting a much higher latency than those sent to a CPU on the local node. Combining the results from figure 7.1a and figure 7.1b enables estimation of just how much more expensive these remote IPIs are. Given that the total number of cycles required to execute the workload is the sum of the cycles spent on IPIs to the local socket, IPIs to a remote socket and a constant representing the remainder of the code, the following holds:

$$cycles = \frac{a \times IPIs}{sockets} + (n \times a) \times IPIs(1 - \frac{1}{sockets}) + C$$

With:

$$a = cycles(IPI_{local})$$

$$n = \frac{cycles(IPI_{remote})}{cycles(IPI_{local})}$$

$$C = cycles(remaining\,code)$$

Substituting C for the amount of cycles used by the benchmark when executed on a single CPU and $IPIs$ for the number of IPIs sent by the 12-CPU variant of the benchmark (both derived from figure 7.1a) allows for determining a and n by curve fitting the above formula to the results from figure 7.1b. This yields a near-perfect fit for $a = 3200$ and $n = 3$, which indicates that IPIs sent to a remote socket are approximately 3 times as expensive as those sent to CPUs on the local socket. The solid lines on figure 7.1b represent the determined curve (adjusted accordingly for execution time).

7.2.3 Summary

Combining the knowledge obtained in chapter 4 with the findings presented above, it is clear that while in basic use cases TLB shootdowns are sufficiently efficient, many factors may drastically increase their cost. What makes this observation so worrying is the fact that the four system properties identified in this dissertation as being detrimental to TLB shootdown performance (high CPU count, NUMA, virtualization and hardware overcommitment) are becoming increasingly prevalent since the rapid rise of cloud computing implies that ever more workloads are being executed in highly consolidated virtualized environments hosted on many-core NUMA systems. This indicates that addressing excessive TLB shootdown overhead is paramount and will likely only increase in importance as time goes on.

7.3 Memory Management & TLB Shootdowns

Evidently, the impact of TLB shootdowns on application performance depends on both the cost of individual shootdowns and the number of these shootdowns an application induces. While the previous section (in combination with knowledge from earlier chapters) already provided great insight into the former, the latter remains mostly unclear. This section addresses this question by

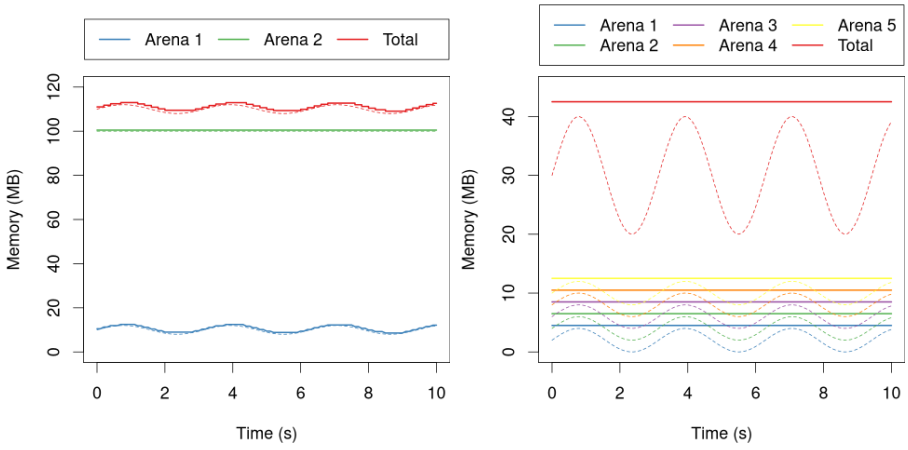
dissecting memory allocator behavior with regard to TLB shootdowns. While today many memory allocators exist with wildly varying implementation details, with respect to system interaction (and thus TLB shootdowns) only a handful of principles are commonly applied. Each of these is detailed below.

7.3.1 Hysteresis-Based Arenas

Early memory allocators were poorly scalable since they serialized all heap modifications by means of a global lock. As heavily multithreaded applications became more common, allocators started dividing this monolithic heap structure into multiple smaller, strictly isolated autonomous arenas, each protected by their own lock, in an effort to alleviate thread contention. However, this meant that the need for interaction with the OS in order to expand or shrink the application's memory space had to become more frequent and fine-grained so as to limit the fragmentation issues inherent to partitioning the memory space. To avoid this OS interaction in turn becoming excessive, hysteresis was employed in the form of padding when the heap is expanded and a trim threshold which must be exceeded before it is shrunk. Many contemporary memory allocators are still based on this concept, most notably glibc's ptmalloc2 [179].

While fine-grained resizing of arenas based on hysteresis is efficient in terms of memory usage, many of these resizing operations require the exact system calls listed in §7.1 as inducing TLB shootdowns. Meanwhile, the global memory efficiency gained by resizing an arena is often minor, as individual arenas may only hold a fraction of the total memory used by the application. There may thus be an imbalance between the rate at which the (relative) memory footprints of individual arenas and that of the application change, suggesting that aggressively resizing arenas based on simple hysteresis thresholds may often not be worth the cost from an application-wide perspective. This dissertation refers to this phenomenon as the 'arena imbalance issue'. Figure 7.2 clarifies this problem visually.

Figure 7.2a simulates the memory footprint of a multithreaded program using two arenas: one with a constant memory consumption of 100 MB and one with a memory consumption oscillating between 8 MB and 12 MB. A simple hysteresis-based memory management algorithm is used with a realistic padding of 500 kB and trim threshold of 1 MB. It is clear that this algorithm maintains excellent memory efficiency, as the actual application memory consumption (dashed lines) and the memory allocated from the OS by the memory manager (solid lines) are hardly distinguishable in figure 7.2a. However, this excellent efficiency comes at a non-negligible cost in performance. Namely, arena 1 is resized 34 times during the 10 seconds of simulated program execution shown



(a) Small threshold. (b) Large threshold.

Figure 7.2: Memory footprints of some hypothetical programs when using hysteresis-based arenas. Solid lines represent the memory allocated by the application while dashed lines represent the memory actually used.

in figure 7.2a. Within the context of arena 1 alone these resizing operations are justified, since the memory footprint of this arena fluctuates by up to 50%. Within the broader context of the application however the story becomes much different, since this same fluctuation only influences the total application memory footprint by 4%. Therefore, it is fair to argue that resizing arena 1 at all is pointless and only incurs unnecessary overhead given its small size compared to arena 2, especially considering the potentially high cost of resizing arenas on modern systems laid bare in §7.2.

Based on figure 7.2a, one may argue that the solution to the arena imbalance issue is simply increasing the hysteresis thresholds. This can even be done dynamically in function of the application's memory allocation behavior. In fact, this is the approach taken by most modern allocators employing hysteresis-based arenas. However, respecting the strict isolation between arenas this paradigm enforces, it is nigh impossible to determine thresholds that perform well for any application. For example, with the benefit of hindsight, reasonable padding and trim thresholds for the program in figure 7.2a would be respectively 2.5 MB and 5 MB. This would eliminate all arena resizing operations, while reducing memory efficiency by only a few percentage points. However, the same thresholds would be catastrophic for a program using a large number of arenas with each a small, though heavily fluctuating demand for memory, as figure 7.2b illustrates. It is

clear that for this program, such enlarged thresholds are not satisfactory since while they do eliminate heap resizing operations, memory efficiency drops to less than 50% for a considerable portion of the program's execution. Thus, the arena imbalance issue is inherent to hysteresis-based arenas and can not easily be resolved by tweaking hysteresis thresholds.

Because each memory allocator employing hysteresis-based arenas uses different thresholds, the exact programs suffering from the arena imbalance issue vary between them. Intuitively however, for any such allocator, programs exist that trigger this problem. Listing 7.2 shows a minimal example of such a program in the case of the most popular memory allocator based on hysteresis-based arenas today: glibc's ptmalloc2.

```
void* work(void* arg)
{
  void* m[1000];
  for (int i = 0; i < 1000; i++)
  {
    for(int j=0;j<1000;j++)
      m[j]=malloc(130048);
    for(int j=0;j<1000;j++)
      free(m[999-j]);
  }
}

int main(int argc, char** argv){
  pthread_t threads[16];

  for (int i = 0; i<16;i++)
    pthread_create(&threads[i], NULL, work, NULL);
  for (int i = 0; i<16;i++)
    pthread_join(threads[i], NULL);

  return 0;
}
```

Listing 7.2: Minimal program suffering from the arena imbalance issue when using ptmalloc2.

In ptmalloc2, the default padding and trimming thresholds are both 128 kB [179]. Listing 7.2 exploits this fact by allocating 1000 chunks of 127 kB of memory, only to deallocate all of them again in reverse order. This process is contained in a loop, which is executed by 16 threads in parallel. On the test system described in §3.3.1, this program induces 230 million TLB shootdown IPIs. Interestingly, if chunks are freed in the same as opposed to reverse order as they are allocated in in listing 7.2, TLB shootdowns and program execution time

are reduced by respectively 99.8% and 87%. This is not particularly surprising, since arenas can not be trimmed when their top chunk is in use. Much more interesting is the observation that the arena imbalance issue can be induced easily for even state-of-the-art memory allocators employing hysteresis-based arenas through seemingly innocuous source code. Moreover, minor changes to said source code may drastically alter the severity of this issue.

7.3.2 Decay-Based Purging

While hysteresis is the most commonly used method to combat excessive resizing operations for memory allocators employing arenas, alternative approaches exist. The most prevalent of these is called 'decay-based purging'. Rather than evaluating if the amount of free memory at the top of the heap exceeds a threshold upon every free operation, the freed memory is gradually released to the OS after a set amount of real time has elapsed (typically seconds). The most popular memory allocator based on this principle is FreeBSD's jemalloc [200].

While decay-based purging intuitively largely mitigates the arena imbalance issue, it introduces a capacitive effect to application memory usage. Particularly for applications with a rapidly and heavily varying memory footprint throughout their execution, decay-based purging is significantly less efficient than hysteresis-based trimming. Figure 7.3 shows an example of such an application.

The application figure 7.3 simulates has a base memory usage of 10 MB, which occasionally briefly peaks to 100 MB. Because these peaks are so sparse however, the average amount of memory the application requires during the 10 second interval shown in figure 7.3 is only 28 MB. However, due to the capacitive effect of decay-based purging, the memory released after each peak is never returned to the system since the time interval between peaks is shorter than the decay time threshold the simulated allocator employs, being 10 seconds (which mirrors jemalloc). Therefore it is clear that decay-based purging is not likely to induce many TLB shootdowns, albeit at a considerable cost in memory efficiency for particular applications.

7.3.3 Size Class-Based Memory Management

Because of the issues associated with arena-based memory management outlined above, some memory allocators opt not to use arenas at all. Instead, they employ thread-local caches which consist of a series of bins, each containing a list of chunks of a fixed size class. Each allocation request is assigned a size class

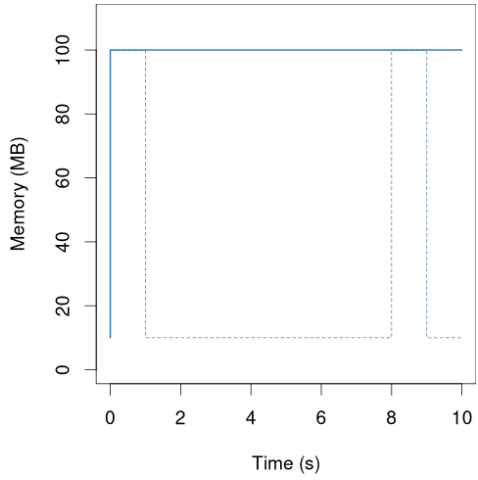


Figure 7.3: Memory footprint of a problematic application for memory allocators employing decay-based purging. The solid line represents the memory reserved from the OS while the dashed line represents the memory actually being used.

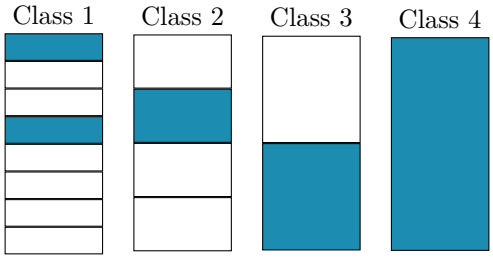


Figure 7.4: Schematic representation of a thread-local cache. Blue blocks represent allocated memory. White ones represent free space.

based on its size and directly served from the corresponding bin. If necessary, each of these bins may be replenished in batches from a central heap. Figure 7.4 illustrates what such a thread-local cache looks like.

Figure 7.4 immediately reveals the principal drawback of size class-based memory allocation, namely fragmentation. The main driving factor behind this undesirable side effect is the fact that freed chunks can only be recycled by allocations corresponding to the same size class [201]. Therefore, the allocator must often request more memory from the system to satisfy an allocation of a particular size class while bins pertaining to other size classes have plenty

of free space to serve the request. Especially for long-running programs which tend to have a sparsely populated memory space, this may lead to abysmal memory efficiency. For example, the simulated program in figure 7.4 requires 4 bins, even though the program only needs 2 bins worth of memory as the chunks assigned to size classes 2 and 3 would have easily fit in size class 1's bin, freeing up bin 2 and 3 for return to the OS.

In spite of the fragmentation issues innate to size classes, many applications make use of this memory allocation paradigm. Namely, allocators based on size classes offer excellent performance and near-infinite scalability due to a lack of thread contention and the absence of complex free list traversal algorithms. Examples of such performance-oriented allocators include tcmalloc [178] and memcached [202].

7.3.4 Garbage Collection

Today, most memory allocators are based on the principle of garbage collection. In contrast to the mechanisms described above, these allocators perform memory management entirely algorithmically, without programmer intervention. The garbage collection algorithm itself works as follows: the algorithm starts at certain predetermined root points, such as active threads and static and local variables. Next, the algorithm identifies all the objects referenced by these root points, which are considered to be active. It repeats this reference tracking process for each identified active object until it can no longer find more of them. All objects which can not be reached through this process are considered garbage. The memory associated with them is thus eligible for return to the system. Figure 7.5 represents this algorithm schematically.

Figure 7.5 suggests that garbage collection is an expensive process, since a large portion of the memory space of the application must be traversed before any garbage may be identified. In fact, programming languages often defer garbage collection as long as possible due to its prohibitive performance impact. For example, in Java, a large amount of memory is reserved when the program starts and the garbage collector is only executed when all of this memory has been allocated, irrespective of how much of it has become garbage in the meantime [203]. While this deferment may minimize performance overhead (including TLB shootdowns), memory efficiency clearly suffers greatly since heap sizes are altered only sporadically and coarsely. In fact, many studies have found that garbage collection has a detrimental impact on memory efficiency [204, 205, 206]. Thus, when efficiency is a concern, allocators employing garbage collection are not an option. Examples of memory allocators employing garbage collection include those used by Java, .NET, Python, etc.

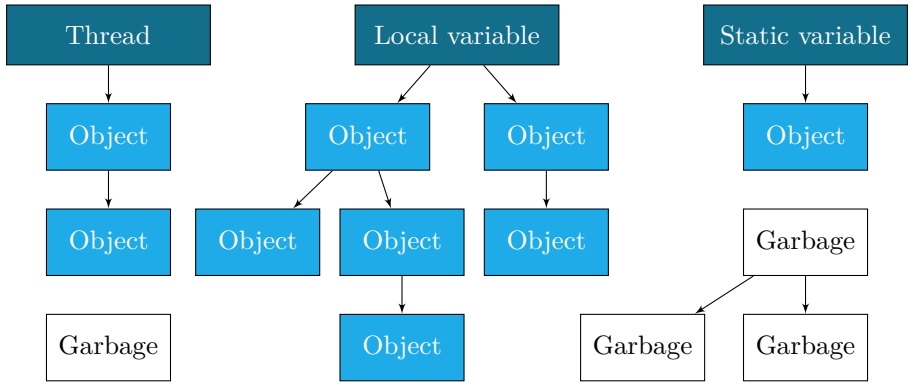


Figure 7.5: Schematic representation of the garbage collection algorithm.

7.3.5 Summary

The above suggests that excessive TLB shootdown overhead is not much of a concern, since many of the discussed memory allocation paradigms do not tend to resize the heap excessively and therefore are not susceptible to high TLB shootdown costs. However, as noted in §5.3.5, all of these paradigms achieve this as a side effect of sacrificing memory efficiency in favor of other design goals, such as overall performance or scalability. One may argue that such design decisions are justified, since memory has become abundant and cheap. However, the widespread adoption of server consolidation demands reconsideration of this argument. Namely, in heavily consolidated environments (e.g. public clouds) increased memory efficiency directly translates into a system being able to host more applications. This especially holds true in containerized environments, since containers are so lightweight that their memory footprint is largely determined by the applications they are hosting [207]. Moreover, increased application memory efficiency often has a positive effect on the invoice of public cloud consumers.

This section has made clear that the only truly efficient memory allocation paradigm—to the best knowledge of to the author—is hysteresis-based arenas. However, this paradigm suffers from the arena imbalance issue, which results in high TLB shootdown overhead. As such, currently it appears that no memory allocation paradigm exists that combines excellent memory efficiency with minimal TLB shootdown overhead. The increasing cost of TLB shootdowns detailed in §7.2 combined with the returning need for highly efficient application runtime environments driven by modern consolidated platforms nevertheless makes a convincing case for such a paradigm.

7.4 Global Hysteresis

As stated in §5.3.5, this dissertation aims to address the observation concluding the previous section by introducing the concept of global hysteresis. As the name suggests, this concept is based on hysteresis-based arenas, mainly because this facilitates interpreting and implementing global hysteresis as an extension to particular existing allocators, which would in turn allow global hysteresis to have an immediate impact in industry. Concretely, this means that formulating global hysteresis equates to formulating a method to eliminate the arena imbalance issue in hysteresis-based arenas (see §7.3.1).

Because the root cause of the arena imbalance issue is an excess of fine-grained arena resizing operations which—despite from the perspective of a single arena appearing appropriate—have no significant impact on the aggregate memory footprint of the application, mitigating it requires answering the following question whenever an arena appears to be in need of resizing:

Does the change to the memory footprint of the application justify the performance overhead of resizing the arena?

Answering the above question requires knowing the benefits of a pending arena resizing operation regarding the application's memory footprint on the one hand and the cost of the resizing operation—which is dominated by the cost of the TLB shootdown it potentially induces—on the other. The memory allocator may then balance these factors as it sees fit, potentially allowing for low memory efficiency in individual arenas when the global impact thereof is minor relative to the cost of resizing said arenas. Note that this mandates a global notion of the application state, in contrast to classic hysteresis which only ever considers the state of the local arena. In other words, implementing global hysteresis as an extension to hysteresis-based arenas requires partially breaking the strict isolation between arenas, allowing basic usage statistics to be exchanged between them in order to determine whether the benefits of a potential arena resizing operation outweigh its cost from the perspective of the application as a whole.

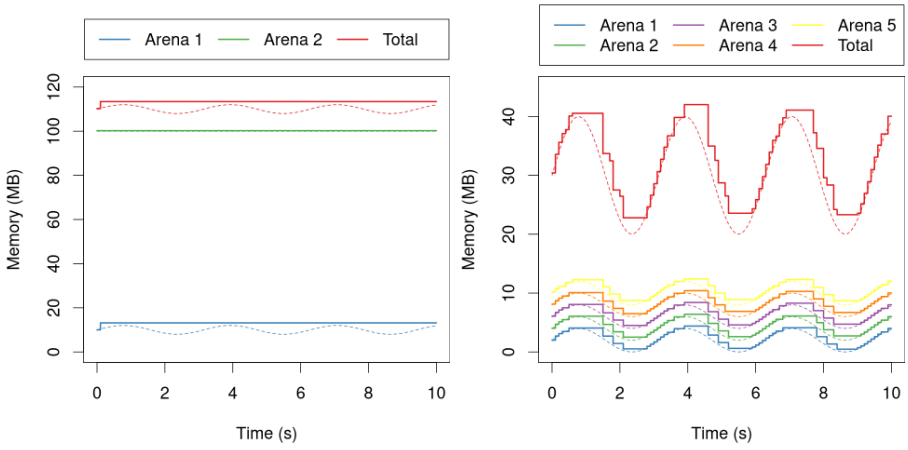
The first challenge to answering the question above concretely is estimating the cost of a TLB shootdown. Sections 4.2 and 7.2 indicated that this cost mainly depends on three factors: the number of CPUs currently being used by the application, how these CPUs are scheduled on the potential NUMA nodes of the system and the presence of virtualization. Many systems allow applications to query these variables at runtime. Thus, the cost of TLB shootdown may be estimated intermittently throughout program execution based on the following formula:

$$(n_{CPU_L} - 1) \times C_{IPI_L} + (n_{CPU} - n_{CPU_L}) \times C_{IPI_R} + V \times C_{exit} \times (n_{CPU} - 1)$$

With n_{CPU_L} the number of CPUs used on the local NUMA node, C_{IPI_L} and C_{IPI_R} the number of cycles needed for sending in IPI to a CPU on respectively the local or a remote NUMA node, n_{CPU} the total number of CPUs used by the application, V a value of 1 or 0 depending on whether or not the system is being virtualized and C_{exit} the number of cycles required for processing an ICR MSR write VM exit. Note however that C_{IPI_L}, C_{IPI_R} and C_{exit} are to be statically and heuristically determined and may vary strongly between systems. Therefore, the practical value of the above formula may be disputed. Moreover, even if the exact cost of an arena resizing operation is known, balancing this cost with memory efficiency remains a heuristical matter which should ideally be determined on a per-use-case basis. Therefore, while some dynamically determined notion of TLB shootdown cost is central to global hysteresis, it refrains from specifying exactly how this cost should be calculated, nor how it should be balanced with memory efficiency. Instead, it leaves these matters to specific implementations, albeit strongly suggesting the use of some incarnation of the above formula. Additionally, it is prudent to provide application developers the option to finetune this cost calculation mechanism, essentially allowing them to specify to what extent the memory allocator should value performance relative to memory efficiency.

Besides TLB shootdown cost, global hysteresis requires knowledge of the memory footprint of the entire application. This can be achieved by simply iterating over all arenas and calculating their cumulative memory usage on a regular basis (e.g. upon each memory allocation or deallocation). Based on this, suitable global top padding and trim thresholds can be heuristically determined as a percentage of the total application memory usage. These thresholds should then be finetuned based on the factors discussed above (TLB shootdown cost and programmer preference). The resulting thresholds may then be applied in the following manner:

- Whenever an arena must be expanded, the local amount of top padding to be added is the global top padding threshold divided by the number of arenas;
- Arenas are trimmed whenever the total free top space of the application exceeds the global trim threshold.



(a) Application from figure 7.2a.

(b) Application from figure 7.2b.

Figure 7.6: Memory footprints of the hypothetical programs from figure 7.2 when using global hysteresis. Solid lines represent the memory allocated by the application while dashed lines represent the memory actually used.

The exact weights to be used in determining the global padding and trim thresholds from application memory usage, as well as the specifics of when they are calculated and how arenas are resized depend on developer preference, application domain and the target system. All of these are therefore left to the implementation.

The above makes clear that the underlying mechanism by which global hysteresis aims to eliminate the arena imbalance issue is allowing for much more free top space in individual arenas than traditional hysteresis-based approaches, as long as the memory usage of said arenas is small with respect to the memory usage of the entire application. When this is not the case, global hysteresis behaves much like hysteresis-based arenas. To demonstrate this concretely, figure 7.6 shows the simulated memory footprints of the applications introduced in figure 7.2 if their memory allocators were to be based on global hysteresis rather than hysteresis-based arenas. The global top padding and trim thresholds were set to 5% and 10%, respectively.

Figure 7.6a shows promising results. Because the variance in arena 1 only has a minor effect on application memory usage, it is hardly ever resized. In total, the application only induces 3 arena resizing operations using global hysteresis, compared to 34 using hysteresis-based arenas. Figure 7.6b on the other hand shows that global hysteresis is also capable of handling scenarios in

which individual arenas do significantly affect application memory usage. In this example, global hysteresis performs nearly identically to hysteresis-based arenas with a realistic hysteresis threshold. Namely, if the same threshold used in figure 7.2a were to be applied to figure 7.2b, hysteresis-based arenas would induce 170 arena resizing operations for this workload, albeit resulting in excellent memory efficiency. Global hysteresis on the other hand induces 198 arena resizing operations, achieving comparable memory efficiency. Thus, with respect to hysteresis-based arenas, global hysteresis effectively mitigates the arena imbalance issue at the cost of some thread contention.

7.5 Implementing Global Hysteresis

Analogously to chapter 6, this chapter aims to exceed purely theoretical contributions through providing a functional prototype implementation of the concept it proposes. As such, this section introduces ptlbmalloc2: an implementation of global hysteresis developed as an open-source library[1] on top of ptmalloc2. The latter was chosen as a baseline because it is a widely used, open source and well documented memory allocator based on hysteresis-based arenas. As stated in §7.4, this incremental approach allows ptlbmalloc2 to make an immediate real-world impact by allowing existing projects to easily integrate it into their codebase. Appendix B lists all of the ptlbmalloc2 source code.

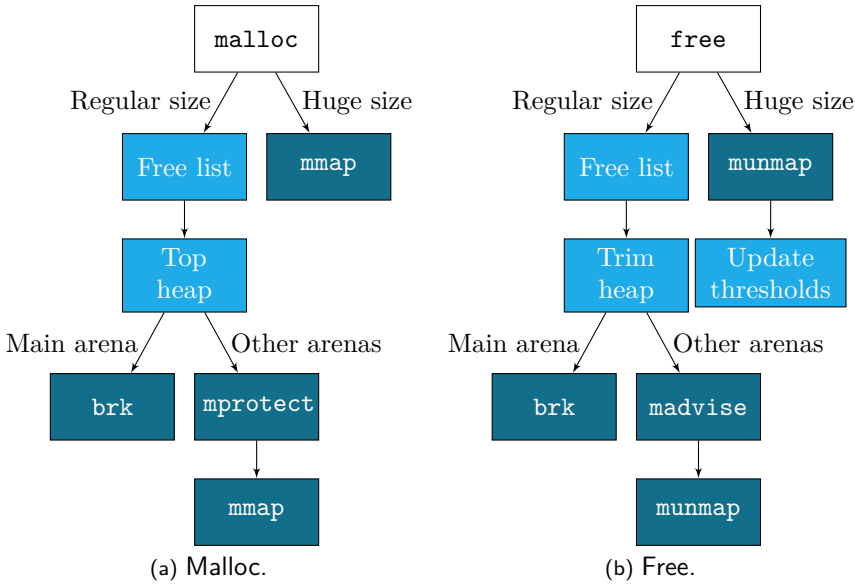
Implementing any library as an extension to existing code requires intimate knowledge of the latter so that the original and novel components interact harmoniously. Therefore, this section first provides an overview of the implementation of ptmalloc2, after which it details how specific aspects thereof were altered in order to implement global hysteresis, giving rise to ptlbmalloc2.

7.5.1 Ptmalloc2

Figure 7.7 provides a simplified schematic overview of the workings of the `malloc` and `free` functions of ptmalloc2, which are the main routines used to respectively allocate and free memory [179].

Ptmalloc2 most often allocates a dedicated arena for each application thread. These arenas may consist of multiple contiguous memory regions, somewhat confusingly called heaps. As figure 7.7a shows, large `malloc` calls are served directly using the `mmap` system call. For smaller chunks, a variety of bins is traversed in search of a suitable previously freed block. If this search is fruitless,

[1] `https://github.com/StijnSchildermans/tlb_shootdown_mitigation.git`



(a) Malloc. (b) Free.

Figure 7.7: Simplified schematic overview of the main routines of ptmalloc2.

the block is allocated from the top of the arena, if sufficient free space is available. If not, the arena is expanded first. For the main arena, this expansion happens through the `brk` system call. For other arenas, the process is slightly more complicated. If the current heap can still be expanded, `mprotect` is used to mark the pages just above the current top of the arena as readable and writable, which allows the application to access them. If not, a new heap is added using `mmap`, with all page permissions disabled before calling `mprotect` to make part of it usable. In this way, newly allocated heaps are not backed by physical memory before they are actually needed by the application. Only in case of the main arena, top padding is included upon expansion.

Figure 7.7b shows how ptmalloc2 handles `free` calls. Namely, when the freed chunk is sufficiently large, it is immediately returned to the system using `munmap` and the hysteresis thresholds are updated based on the size of the chunk. Smaller chunks are added to one of the free lists. Next, the arena is trimmed if its top space exceeds the trim threshold, leaving a small amount of padding. Again, for the main arena this process differs from the other arenas. The former is trimmed using `brk`, while for the latter `madvise(MADV_DONTNEED)` is used. If this `madvise` call transgresses heap boundaries, the topmost one (which now no longer holds any allocated chunks) is returned to the system using `munmap`.

Note that `brk`, `munmap`, `mprotect` and `madvise` all induce TLB shootdowns, as described in §7.1. This causes the overhead associated with the arena imbalance issue described in §7.3.1.

Ptmalloc2 is able to function as described above by using several data structures to track the state of chunks, heaps and arenas. Conveniently, the addresses of these data structures may be derived from the pointer value returned by `malloc`. Effectively, knowing any chunk pointer thus allows for querying the state of the entire memory space. Another interesting side node is the fact that one is able to tune ptmalloc2's behavior at runtime through the `mallopt` routine. This function grants control over padding and trim thresholds, the minimum size of chunks to be allocated using `mmap`, etc. [208].

7.5.2 Ptlbmalloc2

The above has made clear that ptmalloc2 provides all the tools necessary to implement ptlbmalloc2 as envisioned above; a library on top of ptmalloc2 which can be linked into any existing application. It is even possible to recycle much of ptmalloc2's code using the following approach:

1. Define ptlbmalloc2's API as an identical copy of that of ptmalloc2 and perform the necessary linker configuration so that any application calls to ptmalloc2 routines now point to ptlbmalloc2;
2. Disable ptmalloc2's heap trimming and top padding upon the first call to ptlbmalloc2's routines using `mallopt`. Ptmalloc2 is now technically no longer using hysteresis-based arenas;
3. Forward any call to ptlbmalloc2 internally to ptmalloc2 and obtain a pointer to the latter's internal data structures from either the return value of the forwarded call (`malloc`, `calloc`, etc.) or the parameters passed by the caller (`free`, `realloc`, etc.);
4. Using the obtained pointer, determine the state of the entire application memory space whenever prudent;
5. Apply the principles of global hysteresis on the obtained data to determine appropriate padding and trimming thresholds;
6. When necessary, resize arenas using the appropriate system call and update ptmalloc2's internal data structures accordingly using the previously obtained pointer.

Memory consumption	Base threshold
0B - 500 kB	100 kB
500 kB - 1 MB	50%
1 MB - 1 GB	10% + 400 kB
1 GB - ∞	100 MB

Table 7.1: Base thresholds used by ptlbmalloc2 in function of total application memory consumption.

The above constitutes the basic implementation of ptlbmalloc2. The only remaining question is how exactly ptlbmalloc2 approaches the aforementioned 'principles of global hysteresis'. Specifically, three aspects thereof warrant detailed explanation: threshold calculation, claiming memory and trimming arenas. The remainder of this section elaborates on each of these in turn.

Threshold Calculation

In order to efficiently calculate global padding and trimming thresholds, ptlbmalloc2 maintains an array containing pointers to all of ptmalloc2's arena data structures. On each `malloc` call, the arena of the newly allocated chunk is added to this data structure, if it was not already present. On each free call, ptlbmalloc2 calls ptmalloc2's `free` routine and checks if this has changed the size of the arena the chunk belonged to significantly. If so, it calculates the cumulative memory usage of all arenas by iterating over its array of arena pointers. Note that this requires briefly locking each arena to query its metadata. Based on the obtained data, ptlbmalloc2 heuristically determines a base global threshold value. Table 7.1 shows the precise magnitude of this base threshold in function of application memory consumption.

Besides application memory usage, §7.4 prescribes taking the cost of TLB shootdowns into account when determining thresholds. Because this cost is not easily determined, the author judged that for a proof-of-concept implementation of global hysteresis a limited implementation of this aspect suffices. Specifically, ptlbmalloc2 only takes the amount of CPUs currently used by the application into account. It does this by programming a periodic interrupt at a rate of 1 Hz performing the `times` system call, which yields the total CPU time used by the program. Based on this, ptlbmalloc2 determines the average number of CPUs used in the past second. The base threshold from table 7.1 is then multiplied by a heuristically determined factor of $1 + \frac{CPUs}{100}$. This yields the global trimming threshold used by ptlbmalloc2. The global top padding threshold is set to 25% of this value.

Claiming Memory

In contrast to ptmalloc2, ptlbmalloc2 preemptively applies top padding to all arenas. It determines the amount thereof by dividing the global padding threshold by the number of arenas. After every `malloc` call, ptlbmalloc2 determines if the usable top space of the arena is at least 25% of this value. If not and the heap can still be expanded, it calls `mprotect` to set the desired top padding. Finally, it updates ptmalloc2's internal data structures to be consistent with these changes.

Trimming Arenas

Whenever ptlbmalloc2 iterates over arenas to determine hysteresis thresholds, it also calculates the cumulative free top space. If this value exceeds the global trim threshold, it trims all arenas who's top space exceeds twice the per-arena top padding threshold so that the free top space of that arena equals said per-arena top padding threshold. In keeping with ptmalloc2, ptlbmalloc2 trims the main arena using the built-in glibc function `malloc_trim` (which employs `brk` internally) and all other arenas using `madvise`.

7.6 Evaluation

Much like the main purpose of implementing paratick was providing evidence for the efficacy of virtual scheduler ticks in chapter 6, within the context of this dissertation the main purpose of implementing ptlbmalloc2 is assessment of the efficacy of global hysteresis. Following the template of chapter 6, this section presents an evaluation of ptlbmalloc2 based on controlled experiments set up according to the prescriptions provided in §3.3. The employed OS is Ubuntu 20.04 and the assessed workloads are the PARSEC 3.0 benchmarks.

Because ptlbmalloc2 is based on and integrated tightly with ptmalloc2, all results shown in this section represent the performance of ptlbmalloc2 relative to that of ptmalloc2. A large body of existing work in turn compares the latter to other memory allocators, facilitating extrapolation of the results presented here [209, 201]. Note that much like §6.5, this section does not present results in terms of $\delta\eta_r$ and $\delta\eta_t$ since a more direct comparison to existing technologies provides better insight into the performance implications of ptlbmalloc2. Moreover, as §7.2 has indicated that ptlbmalloc2 provides benefits to workloads in native and virtualized scenarios alike, it is prudent to employ virtualization-agnostic performance metrics for its evaluation. Concretely, the following were chosen:

- **TLB shootdowns**: This measure allows for evaluating to what extent ptlbmalloc2 achieves its principal goal, namely eliminating excessive TLB shootdown overhead via mitigating the arena imbalance issue;
- **Memory efficiency**: Mitigating the arena imbalance issue implies reducing the number of arena resizing operations in certain scenarios, which in turn implies that memory efficiency may suffer in favor of improved performance. Because ptlbmalloc2 was carefully designed to minimally affect other aspects of ptmalloc2, reduced memory efficiency compared to ptmalloc2 is intuitively its main potential negative side effect, which warrants careful assessment thereof;
- **System throughput**: This metric allows for assessing to what extent any reduction in TLB shootdowns translates to improved system performance. Refer to §6.5 for a detailed description of the implications of system throughput and its measurement;
- **Application execution time**: This dissertation has by now made amply clear that in the case of multithreaded applications, improved system throughput does not necessarily translate to improved application performance. Therefore, application execution time is measured independently. Again refer to §6.5 for a more detailed motivation for employing this metric.

The remainder of this section evaluates ptlbmalloc2 step by step according to the prescription laid out above.

7.6.1 Conceptual Effectiveness

Figure 7.8 displays the conceptual effectiveness of ptlbmalloc2 in terms of TLB shootdowns. The experiments have been conducted in the absence of virtualization, using 16 CPUs on a single NUMA node. Note that because of the large variance in results, this figure employs a logarithmic scale.

Figure 7.8 indicates that for most benchmarks, the number of TLB shootdowns is low, even when using ptmalloc2. This is to be expected, since—as detailed in §7.3.1—only specific memory allocation patterns induce the arena imbalance issue. *Dedup* and *Vips* likely do exhibit such a pattern, given that these benchmarks incur vastly more TLB shootdowns than their peers in figure 7.8. Analysis of the system calls these benchmarks perform reveals that they induce vast numbers of `madvise(MADV_DONTNEED)` calls, which is as stated in §7.5.1 the main system call ptmalloc2 uses to trim arenas.

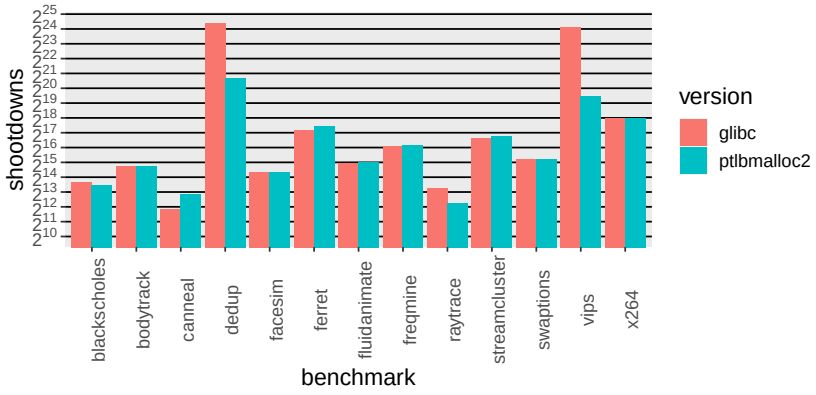
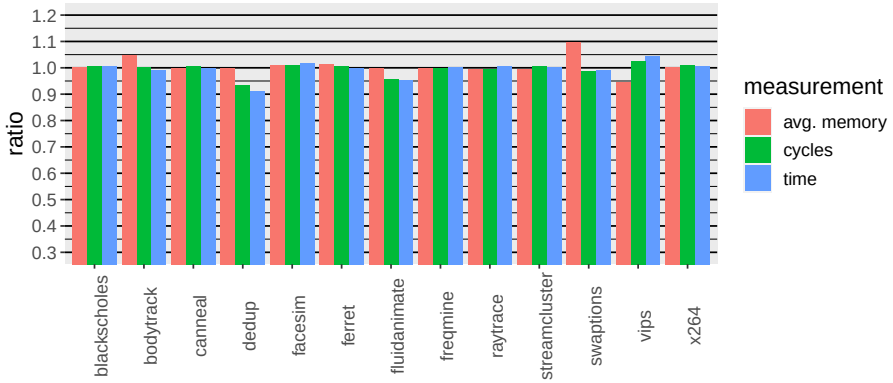


Figure 7.8: Comparison of TLB shootdowns for the PARSEC benchmarks using ptmalloc2 and ptlbmalloc2; run natively with 16 CPUs on 1 socket.

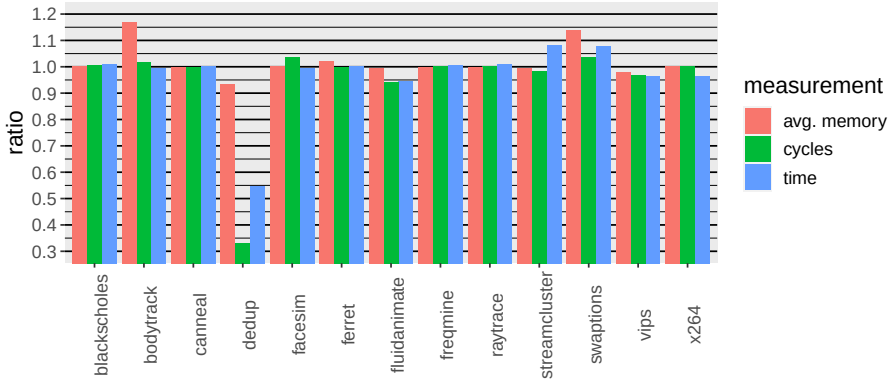
Pertaining to the effectiveness of ptlbmalloc2, figure 7.8 is highly optimistic. Ptlbmalloc2 appears to eliminate almost all TLB shootdowns for the problematic benchmarks without significantly affecting others. This indicates that ptlbmalloc2 achieves its main goal and by extension that global hysteresis is a viable concept.

7.6.2 Side Effects

While §7.6.1 indicates that ptlbmalloc2 is successful at mitigating the arena imbalance issue, it is still unclear whether this comes at the cost of undesirable side effects such as increased resource usage or reduced memory efficiency. To gain insight into this, we next analyze the metrics other than TLB shootdowns outlined in the beginning of this section, which are three of the most important memory allocator performance measures. In the interest of completeness, this analysis includes system configurations with CPU counts varying from 4 to 64, spread over 1 to 4 NUMA nodes. It is limited to native settings only, since §3.2.7 implies that ptlbmalloc2's performance benefits are likely higher in virtualized scenarios due to the various forms of virtualization overhead TLB shootdowns induce. Therefore, potential negative side effects of ptlbmalloc2 are likely more pronounced in a native setting. Figure 7.9 shows the results at the extremes of the spectrum of studied system settings. Other configurations reliably yield results in between these values.



(a) 4 CPUs/threads, 1 socket



(b) 64 CPUs/threads, 4 sockets

Figure 7.9: Average memory usage, execution time and cycles for the PARSEC benchmarks using ptlbmalloc2 relative to ptmalloc2 in various scenarios.

In general, the results in figure 7.9 align with expectations. For most benchmarks, ptlbmalloc2 performs very comparably to ptmalloc2. No benchmark suffers a consistent significant performance degradation across system configurations using ptlbmalloc2. Moreover, careful analysis of the few benchmarks exhibiting a mild slowdown using ptlbmalloc2 in figure 7.9 reveals that the main cause of this performance degradation is increased thread contention for arenas. This occurs partly by design as explained in §7.4 and partly because ptlbmalloc2 is built on top of glibc, rather than as an integrated component thereof. This forces ptlbmalloc2 to contend with ptmalloc2 code for arena locks. While this external approach was a deliberate design decision, direct integration with glibc would likely eliminate the majority of the performance degradation observed in figure 7.9 at the cost of reduced flexibility.

Figure 7.9b indicates that both *Dedup* and *Vips*, which are the benchmarks most likely to benefit significantly from global hysteresis according to figure 7.8, indeed exhibit greatly improved performance using ptlbmalloc2. In figure 7.9a however, *Vips* requires slightly more cycles and time when employing ptlbmalloc2, while strangely memory consumption is 5% lower. This is possible when a benchmark allocates many large chunks. Namely, in ptmalloc2, the trim threshold continues to increase as the `mmap` threshold increases. Ptlbmalloc2 on the other hand bases its thresholds on the application state and may shrink them accordingly. This means that in circumstances of frequent coarse memory allocations, ptlbmalloc2 can be more memory efficient than glibc at a minor cost in performance.

Curiously, *Fluidanimate* consistently shows a performance improvement of ± 5% using ptlbmalloc2 despite figure 7.8 not indicating that this benchmark suffers from the arena imbalance issue. Closer analysis reveals that that this is not a direct consequence of the design considerations of ptlbmalloc2, as the number of system calls performed by this benchmark is identical for ptlbmalloc2 and ptmalloc2. Rather, improved cache performance causes this result. Because cache behavior is very complicated, not a focus of ptlbmalloc2's design and out of scope of this dissertation, it is not fitting to derive any conclusions from this finding.

Despite the design of global hysteresis often allowing for significantly larger hysteresis thresholds for individual arenas than traditional hysteresis-based arenas, figure 7.9 indicates that ptlbmalloc2's memory efficiency is comparable to that of ptmalloc2. Only *Bodytrack* and *Swaptions* show a notable increase in memory usage, which never exceeds 15%. However, analysis of the memory profile of these benchmarks brings to light that they consume very little memory (30 MB for *Bodytrack* and 4 MB for *Swaptions*) to begin with. These results are therefore certainly acceptable.

7.6.3 Performance

Having established that ptlbmalloc2 mitigates the arena imbalance issue without introducing significant side effects, the performance improvement it yields over traditional techniques may be quantified. In keeping with previous chapters, this 'performance improvement' is expressed here in terms of both system throughput and application execution time.

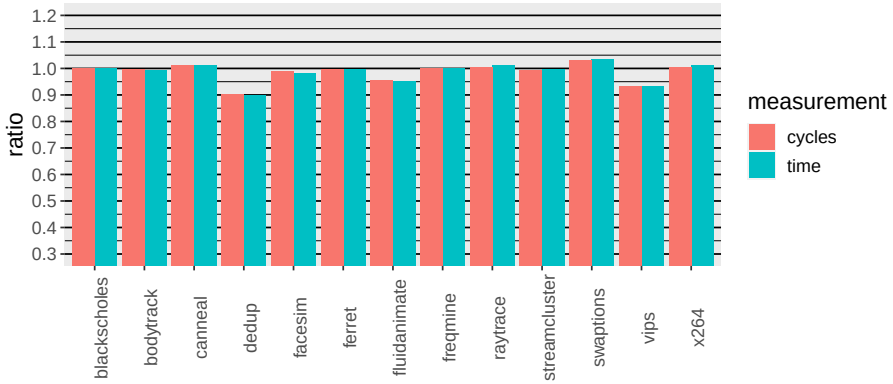
Because both metrics of interest here have already been assessed for native scenarios in §7.6.2, this section only provides a detailed breakdown thereof in virtualized settings. Apart from the presence of virtualization, all assessed system configurations are identical to those assessed in §7.6.2. Figure 7.10 summarizes the results of this evaluation analogously to figure 7.9.

As expected, figure 7.10 shows that the performance improvements yielded by ptlbmalloc2 are greater in virtualized scenarios, in particular for the benchmarks suffering from the arena imbalance issue. All other benchmarks perform nearly identically using either ptlbmalloc2 or ptmalloc2.

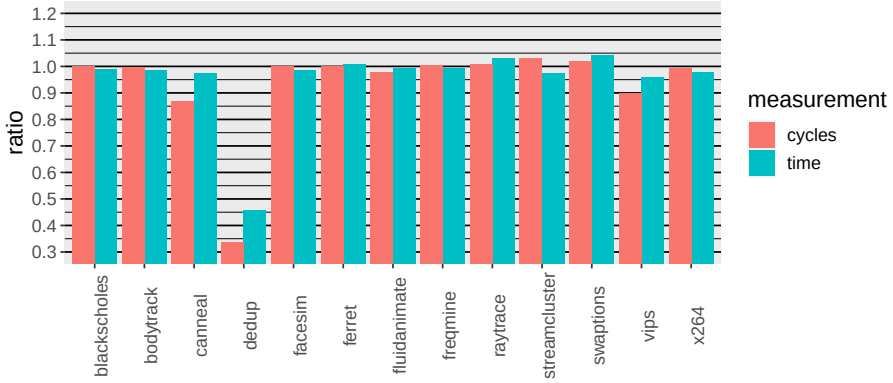
Much like in figure 7.9, several outliers for which ptlbmalloc2 unexpectedly performs notably better than ptmalloc2 may be observed in figure 7.10 (e.g. *Canneal* in figure 7.10b). Further investigation into these outliers reveals that certain benchmarks induce many TLB shootdowns only in specific circumstances. This underlines that memory allocator performance is highly susceptible to platform specifics and that these platform specifics may affect different allocators differently. This in turn implies that limited relative performance variations between ptlbmalloc2 and ptmalloc2 in either direction are unavoidable.

Distilling the results from figures 7.9 and 7.10 shows that ptlbmalloc2 greatly improves performance for benchmarks suffering from the arena imbalance issue relative to ptmalloc2. Most other benchmarks behave nearly identically for both of these allocators, with minor exceptions in both directions. To gain a conclusive insight into the performance of ptlbmalloc2, table 7.2 summarizes the performance of ptlbmalloc2 relative to ptmalloc2 as the aggregate of all PARSEC benchmarks for each of the studied system configurations.

Table 7.2 confirms that on average, ptlbmalloc2 almost always outperforms ptmalloc2 for computation-intensive multithreaded workloads. This performance improvement rises drastically with CPU count. In virtualized environments, the impact is even greater. To the surprise of the author, NUMA has only a limited effect. Nevertheless, the average of all results in table 7.2 is 3% for both throughput and execution time. Ptlbmalloc2 thus boasts tangible performance improvements in the aggregate, with benchmarks suffering from the arena imbalance issue benefiting greatly, while others are not notably affected.



(a) 4 CPUs/threads, 1 socket



(b) 64 CPUs/threads, 4 sockets

Figure 7.10: Average execution time and cycles for the PARSEC benchmarks using ptlbmalloc2 relative to ptmalloc2 in various virtualized scenarios.

Table 7.2: Average performance improvement of ptlbmalloc2 accross all PARSEC benchmarks in all tested scenarios.

Environment	CPUs	Sockets	Cycles	Time
Native	4	1	0%	-1%
Native	4	4	+1%	-1%
Native	16	1	-1%	-2%
Native	16	4	-2%	-4%
Native	64	4	-5%	-4%
Virtualized	4	1	-1%	-1%
Virtualized	4	4	-2%	-1%
Virtualized	16	1	-4%	-4%
Virtualized	16	4	-4%	-3%
Virtualized	64	4	-7%	-5%

Table 7.3: Performance comparison between ptlbmalloc2 and related optimizations reducing TLB shootdowns.

Study	Level	Cycles	Time
Native			
[210]	hardware	/	-5%
[211]	hardware	-5%	/
[212]	system	/	-2%
[196]	system	/	+1%
ptlbmalloc2	**runtime**	**-1%**	**-2%**
Virtualized			
[117]	system	/	-2%
ptlbmalloc2	**runtime**	**-4%**	**-3%**

7.7 Related Work

Most closely related to the work presented in this chapter is literature directly addressing TLB performance, which is plentiful. However, most studies focus on increasing TLB hit rate or reducing TLB miss latency without explicitly addressing TLB shootdowns [176]. Nonetheless, some work directly targeting the latter does exist. Table 7.3 lists all of said work known to the author, providing the level of the system stack at which it was implemented and the performance gains it achieves. The table includes the same information for ptlbmalloc2 for reference.

Even though none of the related work in table 7.3 provides performance figures for both CPU cycles and execution time (or comparable metrics), some clear patterns emerge. Firstly, all of the related solutions to excessive TLB shootdown overhead were implemented at hardware or system software level. This limits their applicability since hardware-based solutions are very costly to implement and system-based ones are limited in their potential for optimization due to generality concerns. Secondly, ptlbmalloc2 achieves comparable performance gains to related solutions at system level. While techniques at hardware level do perform better overall, their widespread adoption would take many years due to the aforementioned costs involved.

More indirectly linked to this chapter—but therefore not less relevant—are ongoing efforts to develop massively scalable OSs [213, 214]. Such systems view every CPU as a discrete entity running its own microkernel. Any communication between CPUs is explicit. This reduces or even eliminates the need for OS-managed TLB consistency, among many other benefits. Although experimental implementations of these systems exist, there are no signs that any of them are to be adopted on a large scale in the foreseeable future.

Finally, memory allocation remains a vivid research field. Recent attempts to improve on the strengths and mitigate the weaknesses of existing memory allocators are ubiquitous, e.g. with regard to synchronization mechanisms [215] or data locality [216]. These efforts are largely orthogonal to the work presented in this chapter.

7.8 Conclusion

Due to several evolutions in the nature of contemporary computing platforms, TLB shootdown cost is steadily rising. Since for multithreaded applications many of these shootdowns are caused by memory management at application level, optimizing memory allocators is a promising method to address this issue. Existing allocators either exhibit poor TLB performance due to the arena imbalance issue or poor memory efficiency due to a focus on performance. This chapter explored the potential of explicitly focussing on the trade-off between (TLB) scalability and memory efficiency, resulting in a memory allocator design concept and a prototype implementation thereof exhibiting excellent performance in both of these metrics with minimal side effects: respectively global hysteresis and ptlbmalloc2.

While global hysteresis achieves its objectives, it is tightly bound to a specific legacy allocator design concept, namely hysteresis-based arenas. The core issue global hysteresis addresses is however much broader: the trade-off between

memory efficiency and performance. Perhaps the most important conclusion of this chapter is that memory allocators in general must reconsider how they interpret the metric 'performance' and start taking into account traditionally insignificant aspects thereof—such as TLB shootdowns—in response to the evolution of the systems their allocators are deployed on.

7.8.1 Personal Contriburion

This chapter entirely consists of original work by the author of this dissertation. As with all other chapters however, his supervisors provided invaluable guidance and feedback throughout.

7.8.2 Future Work

This chapter provides several incentives for future work:

- As stated above, global hysteresis is but one possible angle from which to approach the issue of balancing memory efficiency and TLB performance. Nothing in this chapter suggests that the opposite avenue, namely improving memory efficiency for any of the memory allocation paradigms exhibiting low TLB shootdown overhead (see §7.3) is not feasible. How to develop an allocator from that perspective and how such an allocator would compare to ptlbmalloc2 are interesting open questions;
- While ptlbmalloc2's implementation as a library on top of glibc allows for rapid dissemination, §7.6.2 noted that this has negative performance implications due to contention between ptmalloc2 and ptlbmalloc2. Therefore, implementing global hysteresis as an integrated component of glibc would likely further improve its performance gains over hysteresis-based arenas and is therefore an interesting direction for future work;
- §7.6.2 noted that ptlbmalloc2 appears to improve cache performance for some workloads. The mechanisms behind this improvement and whether it is coincidental or systematic are however unknown. Given that this observation potentially implies an unintended additional asset of ptlbmalloc2, more thoroughly analyzing the latter's effects on cache behavior is of great interest.

Chapter 8

Application Amelioration: Guidelines to Developers

This chapter was previously published as:
S. Schildermans and K. Aerts. "Towards High-Level Software Approaches to Reduce Virtualization Overhead for Parallel Applications". In: *2018 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE. 2018, pp. 193–197

After having presented ameliorations to the virtualization process of multi-threaded applications at system level in chapter 6 and runtime environment level in chapter 7, this chapter targets the highest possible level of abstraction, namely application source code. Indeed, chapter 5 suggested multiple times that this is a promising, yet understudied angle from which to approach this issue and even provided an indication of its potential in §5.1.6.

While system software controls how application requests are processed, the application itself determines the number and nature of these requests to begin with. Therefore, almost all virtualization overhead may be prevented through altering application code so that it avoids operations likely to induce significant amounts of said overhead. However, doing so is not trivial due to the complexity of modern virtualization technologies and the unique nature of each application. The principal goal of this chapter is to aid developers in this process through formulating a set of guidelines and best practices.

Chapters 3 and 4 have provided a wealth of information regarding the various forms of virtualization overhead multithreaded applications are susceptible to, as well as the principal causes thereof. However, these chapters often omitted linking said causes directly to application source code. Therefore, the first contribution of this chapter is to detail exactly how certain application source code triggers the system-level phenomena inducing virtualization overhead discussed earlier in this dissertation. Once this link has been clearly established, it proceeds to formulate the aforementioned set of guidelines. Finally, this chapter provides evidence of the efficacy of the guidelines it proposes by applying them to one of the benchmarks shown in chapter 4 to suffer the most in a virtualized environment, namely the *Dedup* benchmark from the PARSEC benchmark suite. It dubs this implementation 'NODedup'; short for 'No-Overhead Dedup'.

While this chapter—like most of the chapters preceeding it—is based on published and peer-reviewed original work by the author of this dissertation, most of the information it provides was not included in said publication. The reason for this is that that publication dates back to the early stages of the Ph. D. project documented here and was limited to providing evidence for the viability of the application-level approach to addressing virtualization overhead alluded to so frequently in chapter 5. It formulated some initial insights regarding the link between application code and virtualization overhead and documented and evaluated NODedup. Through the years of work that led to this dissertation however, the author's knowledge concerning this topic steadily expanded and crystallized. While time constraints have prevented this additional knowledge from being published in its own right, it is included in this chapter so that it may reach individuals interested in such information regardless. As a result, this chapter in essence comprises a compilation of insights regarding mitigating virtualization overhead through intelligent application design the author obtained while working on the Ph. D. project this manuscript documents, reinforced with peer-reviewed and published evidence.

Main Findings & Contributions

- This chapter clarifies the link between application source code and the different forms of virtualization overhead for multithreaded applications;
- This chapter introduces a set of guidelines and best practices aiding software developers in designing their applications in such a way that they minimize virtualization overhead;
- This chapter provides evidence for the efficacy of the guidelines it proposes.

8.1 Background: The Dedup Benchmark

From the introduction to this chapter it is clear that the *Dedup* benchmark from the PARSEC benchmark suite will play a central role in validating the guidelines it proposes. Even though this workload has been featured throughout this dissertation, none of the previous chapters detailed its inner workings, which is nonetheless paramount in order to implement NODedup and as such perform the validation mentioned above. This section rectifies this by elaborating on the anatomy of this benchmark.

The *Dedup* benchmark featured in PARSEC is an implementation of the well-known data deduplication algorithm written in the C programming language. Data deduplication in turn is a data compression algorithm mostly popular for storing large data sets likely to contain a significant amount of repetition, such as periodic system backups and—highly fitting for this dissertation—stores of VM images in cloud environments [217]. It consists of the following steps:

1. Read the input file from disk and coarsely divide it into chunks;
2. Refine each chunk into smaller chunks;
3. For each chunk, identify duplicates using a global hash table;
4. Compress all first occurances of chunks and replace any duplicates by a reference to their first occurance;
5. Write the output to disk.

The *Dedup* benchmark implements the above algorithm as a parallel pipeline based on pthreads. As soon as chunks are created, they pass through the subsequent pipeline stages in no particular order, before being reordered during the final pipeline stage and written to disk. Additionally, *Dedup* creates multiple threads to handle pipeline stages 2, 3 and 4. The developers of the PARSEC benchmark suite recommend each of these stages to be assigned at least as many threads as there are logical CPUs available to the system, so that the scheduler can accurately balance CPU time between them [124]. *Dedup* employs ring buffers [218] between all pipeline stages to store intermediary results. Access to these buffers is serialized using several blocking synchronization constructs.

8.2 Application Code & Virtualization Overhead

To be able to formulate accurate guidelines regarding writing multithreaded application code inducing minimal virtualization overhead, deeply understanding the connection between said code and overhead is paramount. This section aims to provide such an understanding, primarily through a (non-exhaustive) series of examples. It follows the same structure as chapter 5, addressing each of the high-level causes of virtualization overhead for multithreaded applications identified in chapter 4 independently.

8.2.1 Blocking Synchronization

§4.2 indicates that blocking synchronization is with little doubt the most common cause of virtualization overhead for multithreaded applications; not only because of the many aspects thereof that require special care in a virtualized setting but also because of how central this mechanism is to multithreading. Namely, multithreading is hardly possible without at least some coordination mechanism enabling threads to share data harmoniously. Because blocking synchronization is for most use cases the most efficient of these mechanisms, many multithreaded applications rely heavily on it [219].

Blocking synchronization is typically implemented at OS level and exposed to applications through system calls (e.g. `futex` in Linux [175]). While the API facing user space is most often very simple, programming languages have built plethora of synchronization primitives on top. Because these primitives vary wildly in level of abstraction, working principles and usage, it is worth exploring the most prevalent examples thereof. This is done below.

Mutex

The simplest and most explicit implementation of blocking synchronization is the mutex [220]. It is largely a direct extension of the system level blocking synchronization API: an atomic boolean variable which threads must explicitly lock and unlock when respectively entering and exiting a critical section through dedicated library calls. When a thread attempts to lock an already locked mutex, it blocks. When the thread holding the lock releases it, the OS wakes the blocked thread, which may now claim the lock and enter the critical section.

Many low-level imperative programming languages implement mutexes. Listing 8.1 provides an example of the usage of these primitives in C.

```c
#include <pthread.h>;

pthread_mutex_t mutex;

void* work(void* arg)
{
  pthread_mutex_lock(&mutex);
  //CRITICAL SECTION
  pthread_mutex_unlock(&mutex);
}

int main()
{
  pthread_mutex_init(&mutex, NULL);
  work(NULL);
  return 0;
}
```

Listing 8.1: Mutex example in C.

Listing 8.1 makes it clear that mutexes are easily recognizable due to the explicit library calls they require to denote every critical section. This facilitates identification of source code where their use may be problematic in a virtualized context.

Counting Semaphore

Counting semaphores constitute a more flexible alternative to mutexes. Rather than a boolean variable, they employ a counter. The programmer may initialize this counter to an arbitrary positive integer value and threads may atomically increment or decrement it at any time [221]. When the counter reaches zero, any threads attempting to decrement it further block until some other thread increments the counter again.

Counting semaphores are useful for protecting e.g. limited hardware resources to ensure the system is not overwhelmed. Listing 8.2 shows an example of typical semaphore usage in C.

Listing 8.2 indicates that using semaphores is almost identical to using mutexes, which makes them equally easy to identify. Note however that semaphores are by nature slightly less likely to induce problematic levels of virtualization overhead than mutexes, since they often allow for multiple threads to acquire a protected resource before they start blocking any.

```c
#include<semaphore.h>

sem_t semaphore;

void* work(void* arg)
{
  sem_wait(&semaphore);
  //PROTECTED RESOURCE ACCESS
  sem_post(&semaphore);
}

int main()
{
  //Initialize a semaphore which at most 5 threads may hold
    simultaneously.
  sem_init(&semaphore, 0, 5);
  work(NULL);
  return 0;
}
```

Listing 8.2: Counting semaphore example in C.

Condition Variable

Condition variables allow threads to wait for an event by blocking until another thread determines that said event has occurred [222]. In technical terms, a condition variable resembles a queue of blocked threads, which any thread may join at its discretion by calling a specified library function. Any other thread may at any time signal one or more threads in the queue to resume execution through another library function. Listing 8.3 displays an example of this mechanism in C.

```c
#include <pthread.h>

pthread_t worker;
pthread_mutex_t lock;
pthread_cond_t cond;

void* work(void* arg)
{
  pthread_mutex_lock(&lock);
  pthread_cond_wait(&cond, &lock);
  pthread_mutex_unlock(&lock);
}
```

```
int main()
{
  pthread_mutex_init(&lock, NULL);
  pthread_cond_init(&cond, NULL);
  pthread_create(&worker, NULL, work, NULL);

  //EVENT OCCURED
  pthread_mutex_lock(&lock);
  pthread_cond_broadcast(&cond);
  pthread_mutex_unlock(&lock);

  return 0;
}
```

Listing 8.3: Condition variable example in C.

In C, condition variables are significantly more complicated in usage than the previously discussed synchronization mechanisms, as listing 8.3 indicates. It is also clear that condition variables are by nature highly conducive to virtualization overhead because—in contrast to mutexes and semaphores—when a thread calls to wait for a condition variable, it is guaranteed to block (and thus induce overhead). Moreover, condition variables require the use of mutexes internally, further increasing their cost in a virtualized environment. On a positive note though, this source of virtualization overhead is—even more so than mutexes and semaphores—easily identifiable due to its verbosity.

Monitor

Many modern programming languages embed thread safety directly into the most fundamental language constructs in the form of monitors. In abstract terms, a monitor is a serializing structure encapsulating some resource and coordinating thread access to that resource [223]. It is usually implemented as a combination of mutexes and condition variables. Most contemporary object-oriented programming languages implicitly provide each object with such a monitor. However, for performance reasons, this monitor is usually ignored unless the programmer explicitly requests not to do so by means of adding a specific keyword to any class member declaration (e.g. `synchronized` in Java [224] or `lock` in C# [225]). Some languages (e.g. Python) go even further and encapsulate the entire runtime environment in a monitor, effectively implicitly serializing the entire program. Listing 8.4 shows an example of monitor usage in Java.

```
public class BankAccount{

  private int balance;

  public static synchronized void withdraw(int amount){
      balance -= amount;
  }
}
```

Listing 8.4: Example of a Java class using its monitor.

From listing 8.4 it is clear that monitors are much less explicit than the previously discussed synchronization mechanisms. For instance, it is unclear from listing 8.4 exactly what the monitor protects: The method and its local variables? All the objects the method references? The entire class the method belongs to? It is impossible to answer this question without deep knowledge of the programming language used. This makes pinpointing the origin of virtualization overhead induced by these monitors challenging. Moreover, because of their high level of abstraction, monitors—as implemented in most mainstream programming languages—tend to be overprotective. While the previously described synchronization mechanisms allow fine-grained control over critical sections, monitors evince method- object- or even global granularity. Therefore, they tend to block threads and thus induce virtualization overhead (much) more often than semantically necessary.

Implicit Synchronization

Several alternative programming paradigms which have gained much traction in recent years—e.g. functional programming [226] and reactive programming [227]—allow for a declarative approach to multithreading. This quite literally means that programmers simply declare which sections of their code may be executed concurrently by using dedicated syntax. The application runtime environment may then distribute the work that code describes over any number of threads as it sees fit, performing all aspects of thread management and coordination entirely transparently [228]. In some cases, the runtime environment may even fully implicitly identify code segments that lend themselves to parallelization—and execute these segments accordingly. The most prominent example of this implicit multithreading is MatLab [229].

Even though to many developers semi- or fully implicit parallelism undoubtedly sounds appealing, the convenience it brings inherently comes at a hefty price: users must relinquish control over a large part of the application's operational semantics to the runtime environment. While in most cases the latter is

adequately capable of determining an efficient manner of parallelizing application code, the equation changes drastically in a virtualized environment. After all, most of this dissertation is dedicated to a variety of performance issues multithreaded applications may suffer in a virtualized environment which are not yet fully understood by humans—let alone application runtime environments. Consequently, these automated solutions are likely to employ sub-optimal parallelization techniques such as ubiquitous use of blocking synchronization, even where the overhead of doing so is likely to outweigh its advantages. Listing 8.5 clarifies this by expanding on listing 8.4 using Java's functional `streams` API.

```
public class BankAccount{

  private int balance;

  public static synchronized void withdraw(int amount){
      balance -= amount;
  }
  public static void withdrawMany(Stream<Integer> amounts){
    withdraw(amounts.parallel()
                    .reduce(0, Integer::sum));
  }
}
```

Listing 8.5: Example of a Java class employing a parallel stream.

While listing 8.5 highlights the simplicity and elegance of declarative multithreading, it also indicates the challenges it poses with regard to tracing potential sources of virtualization overhead. In particular, while it is clear that the `withdrawMany` method parallelizes its input stream, it is unclear how many threads it creates and how these threads interact. These questions are of particular importance with regard to the reduction operation this method employs (a function merging the entire stream into a single value), which naturally requires extensive exchanging of results and thus synchronization between threads. While a runtime environment can employ all kinds of heuristics to optimize the number of threads used and the associated need for synchronization, the developer is likely in a much better position to make such judgements, being able to take factors external to the application code itself into account (e.g. input stream size, hardware platform, presence of virtualization, etc.). As such, declarative multithreading often leads to wasteful utilization of resources and in virtualized environments even to non-negligible amounts of virtualization overhead which can be very difficult pinpoint.

8.2.2 Spinning Synchronization

Regarding the relationship between application code and virtualization overhead, spinning synchronization is perhaps the most interesting topic of all. Namely, despite spinning synchronization having received massive attention from literature—as discussed in §3.2.5—§4.2.4 has made clear that spinning at user level may induce catastrophic virtualization overhead, even on state-of-the-art platforms. As such, it is certainly worthwhile to investigate how this construct may manifest itself in application source code, as is done below.

Spin Locks

Analogously to blocking synchronization, many programming languages offer spinning synchronization as part of their core SDK. The abstraction through which they expose this feature is most often the spin lock. These spin locks are largely identical in structure and usage to mutexes (see §8.2.1), the main difference being that instead of blocking, threads attempting to acquire a contended lock enter a busy-waiting loop, as explained in §3.2.5. Listing 8.6 provides an example of the usage of such a spin lock.

```
#include <pthread.h>;

pthread_spinlock_t lock;

void* work(void* arg)
{
  pthread_spin_lock(&lock);
  //CRITICAL SECTION
  pthread_spin_unlock(&lock);
}

int main()
{
  pthread_spin_init(&lock, 0);
  work(NULL);
  return 0;
}
```

Listing 8.6: Spin lock usage example in C.

Comparing listing 8.6 to listing 8.1, it is clear that spin locks are—at least in C—identical in usage to mutexes. This holds true for most programming languages due to the similarity of these constructs. Note however that spin locks are used much less often than mutexes since the latter are almost always

much more efficient. Only for very short critical sections it may be beneficial to use spin locks because of the overhead associated with blocking and unblocking threads, which is even in a native setting not entirely negligible.

Custom Spinning Constructs

Even though many programming languages provide abstractions dedicated to spinning synchronization, many applications make use of custom constructs for this purpose. While implementation details may obviously vary significantly, all of these custom constructs are based on the principle of continually attempting to atomically check and set a boolean variable; only proceeding when succesful. Listing 5.1 already provided a generic example of such a custom spinning synchronization mechanism in the form of a basic spin lock. More advanced variants may deviate slightly semantically or incorporate additional features or performance optimizations. Listing 8.7 shows such an advanced user-level spin lock, written in C++ [230].

```
struct spinlock {
  std::atomic<bool> lock_ = {0};

  void lock() noexcept {
    for (;;) {
      if (!lock_.exchange(true, std::memory_order_acquire))
        return;

      while (lock_.load(std::memory_order_relaxed))
        __builtin_ia32_pause();
    }
  }

  void unlock() noexcept {
    lock_.store(false, std::memory_order_release);
  }
}
```

Listing 8.7: Example of an advanced user-level spin lock in C++.

At first glance, listings 5.1 and 8.7 highlight the diversity of custom user-level spinning synchronization constructs, which suggests identifying them in application source code is challenging. A closer look at these listings however reveals that said constructs tend to have a similar structure, which makes identifying them in the application code base in the case of excessive virtualization overhead relatively straightforward nonetheless.

8.2.3 Data Sharing

The principal negative effect of sharing data between threads in a virtualized context is the overhead associated with the TLB shootdowns this inevitably induces. While these TLB shootdowns are in se purely system-level phenomena, chapter 7 has shown that application software can have a dramatic effect on their prevalence and therefore impact on application performance. While that chapter focussed primarily on the runtime environment, §7.3.1 has indicated that in the end, even when using a susceptible runtime environment inattentively designed application source code is the catalyst for excessive TLB shootdown overhead. Studying precisely how this catalysis takes place is certainly worthwhile in the interest of situations where optimizing the runtime environment is not possible or desirable.

Listing 7.2 already provided an example of an application inducing excessive amounts of TLB shootdowns through the arena imbalance issue, which in turn is tightly linked to said application's source code. In a more general sense, heap resizing operations (and thus TLB shootdowns) are likely when threads often allocate or deallocate memory at the top of their arenas. Unfortunately, it is not possible to model this in function of application source code in any general sense because of the widely varying behavior of different memory allocators (see §7.3) as well as the fact that the exact size of memory (de)allocations often dependends on external factors (e.g. an input file, the result of a database query,...). Nevertheless, from the knowledge obtained in chapter 7, it is possible to derive several application behaviors that increase the likelihood of excessive heap resizing operations occurring:

- Frequent small memory (de)allocations;
- Large amounts of consecutive (de)allocations;
- Increasing allocation sizes as program execution progresses;
- Holding on to memory for long periods of time before deallocating;
- Deallocating memory in reverse order with respect to how it was allocated.

While the above memory management behaviors at first glance appear to be quite distinct, they all either gradually rather than abruptly alter the heap size or decrease the probability that memory allocations may be satisfied from the free list, eventually leading to many allocations being stacked at the top of the heap. Listing 8.8 shows a C program exhibiting all of these potentially problematic behaviors simultaneously.

```
int main()
{
  void* mem[1000];
  for (int i = 0; i < 1000; i++)
    mem[i] = malloc(10 * i);

  application_logic(mem);

  for (int i = 0; i < 1000; i++)
    free(mem[1000-i]);
  return 0;
}
```

Listing 8.8: Memory allocation patters leading to excessive TLB shootdown overhead in C.

Listing 8.8 paints a sobering picture of how easy it is to write code inducing problematic levels of TLB shootdowns in standard C. Namely, because all of the allocations are performed consecutively in the very beginning of the `main` method, the allocator can not use previously freed blocks. Moreover, each allocation is larger than any of the preceding ones, meaning that even if some of the preceeding chunks would have been freed by the time the later ones were allocated, the free list would likely not have been able to serve them. Furthermore, even though `mem` is only used within `application_logic`, the program only frees it after completion of this subroutine. This means that any allocations within `application_logic` must be served from the top of the heap as well. Lastly, memory is deallocated in reverse order compared to how it was allocated, constantly growing the top of the heap and therefore inducing trimming operations.

Problematic memory allocation patters similar to listing 8.8 are in practice not at all easy to identify, since real-world allocation patterns are a complex mix between application code, library routines, runtime specifics, system properties and external factors. Combined with the previously stated variance between memory allocators, even armed with the knowledge outlined above, the only reliable way to definitively pinpoint source code inducing concerning amounts of TLB shootdowns is through careful performance profiling.

8.2.4 Non-Uniform Memory Access Locality

The final high-level cause of virtualization overhead for multithreaded applications chapter 5 recognizes is NUMA abstraction, which may drastically reduce memory locality and thereby increase memory latency in virtualized sysems. While this issue is by definition only relevant when the host system sports a NUMA architecture, it may affect many multithreaded applications when this is the case. Specifically, any application that frequently accesses data from multiple threads simultaneously may suffer. Listing 8.9 provides an example of such an application in C.

```
#inlcude<pthread.h>

void* work(void* arg)
{
    char* ptr = (char*) arg;
    *ptr = 'a';
    return NULL;
}

int main()
{
  pthread_t threads[16];
  char* ptr = (char*) malloc(4096);

  for (int i =0; i < 16; i++)
  {
    void* arg = (void*)(ptr + 256 * i);
    pthread_create(threads + i, NULL, work, arg);
  }

  for (int i =0; i < 16; i++)
    pthread_join(threads + i, NULL);

  return 0;
}
```

Listing 8.9: Program exhibiting poor memory locality in C.

The program in listing 8.9 allocates 4 kB of memory, after which it creates 16 threads which each manipulate a different section thereof. Because the entire allocation fits into a single memory page, it is more than likely that this program will exhibit poor memory locality when executed on a NUMA system, especially in virtualized settings. What makes listing 8.9 especially interesting however, is that none of the threads manipulate exactly the same data. Nevertheless, because the OS manages memory at page granularity, accesses to the same

memory page are equivalent to accesses to the same data with respect to this issue. Exactly this is what makes this problem more prevalent and challenging to address than is apparent at first glance. Knowledge of the exact location in memory of the data used by each thread is therefore necessary to identify code causing performance degradation due to NUMA abstraction. Unfortunately however, because the runtime environment often abstracts such details to a large degree, dynamic profiling is likely necessary to obtain said knowledge.

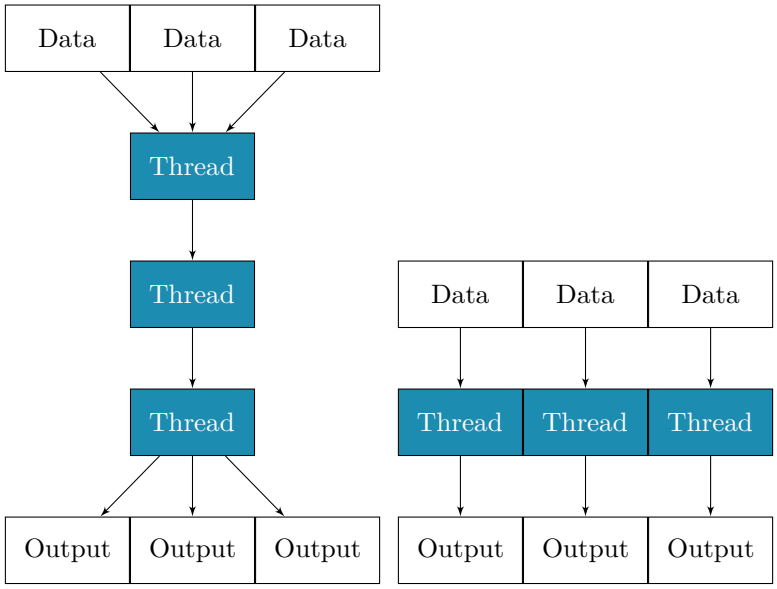
8.3 Guidelines

By describing application code inducing excessive virtualization overhead, the previous section implicitly equally described its antithesis; application code inducing hardly any virtualization overhead at all. This section reformulates this implicit antithesis as an explicit set of guidelines application developers may follow in an effort to minimize the probability that their multithreaded applications will suffer significant virtualization overhead. In practice, these guidelines have been established and refined throughout the Ph. D. project documented in this dissertation and can therefore to some extent be seen as the fruit of all the previously described work. Note that in contrast to the contributions described in previous chapters, these guidelines are not meant to be the infallable gold standard regarding developing multithreaded applications for the cloud. Rather, they are intended as a development aid which minimizes chances of applications incurring high virtualization overhead, albeit without providing any guarantees.

Following the example of the previous section, the aforementioned guidelines are grouped by the high-level cause of virtualization overhead they address and presented accordingly below.

8.3.1 Blocking Synchronization

Since the primary purpose of synchronization in general is guaranteeing correctness by coordinating execution streams, the need for it depends on how these execution streams relate to one another, which in turn largely depends on the application architecture. One important consideration regarding the architecture of any multithreaded application is the approach it takes to the concept of parallelism itself. In this regard, two paradigms exist: data parallelism on the one hand and task parallelism on the other [231]. Figure 8.1 illustrates both schematically.



(a) Task parallelism. (b) Data parallelism.

Figure 8.1: Schematic overview task parallelism and data parallelism.

As figure 8.1a shows, task parallelism equates to dividing a workload into multiple independent tasks and executing these tasks in parallel. This concept is also known as pipelining. Normally, all data is passed through all pipeline stages, each of which is usually associated with an individual worker thread. Data parallelism takes the opposite approach, namely dividing the input data set into independent subsets, each of which is processed entirely by a single thread, as figure 8.1b illustrates.

From figure 8.1, it is evident that data parallelism is to be preferred over task parallelism from the perspective of minimizing the need for synchronization. Namely, task prallelism requires data to be passed between multiple threads, which introduces data dependencies between them. These dependencies in turn call for some form of synchronization—most often blocking synchronization due to its efficiency—to be implemented so that one thread does not access a piece of data before another is finished with it. Data parallelism conversely does not suffer from this issue, since each individual piece of data is handled by exactly one thread.

A further optimization minimizing the amount of synchronization an application requires which naturally combines well with data parallelism is the use of thread pools. A thread pool consists of a centralized set of worker threads (normally limited in size to the number of CPUs available to the application) and a work queue [232]. Application code may at any time submit work to the queue. The worker threads monitor the queue and perform any submitted jobs as soon as possible. If the thread pool is configured appropriately, it minimizes the amount of potential contention for locks and associated overhead through limiting the number of threads and managing these threads using a small set of centralized, highly optimized routines as opposed to ad-hoc application code.

8.3.2 Spinning Synchronization

The best advice possible regarding spinning synchronization at user level is to avoid it at all costs if there is any chance the application may be run in a virtualized environment. The easiest approach to accomplish this is to at all times resort to blocking synchronization or preferably more advanced spin-then-block primitives as suggested in §5.2.4.

8.3.3 Data Sharing

Virtualization overhead caused by data sharing between threads and the ensuing TLB shootdowns is the most challenging form of overhead to address at application level. The main issue here is that the causes of this problem outlined in §8.2.3 are so varied and nuanced that straightforward approaches to addressing some of them may increase the severity of others. For example, one may trivially address the issue of 'many small allocations' listed in §8.2.3 by merging multiple small memory allocations into a single larger one. However, this would likely require holding on to the memory for a much longer time, since this 'superblock' can only be released once the application is finished with all of its previously independed constituents. This prolonged memory retention is in itself listed in §8.2.3 as a cause of excessive TLB shootdown overhead. Moreover, such an approach is likely to non-negligibly reduce memory efficiency and complicate using the memory in question due to additonal addressing abstractions. Such side effects make no single approach to the problem of excessive TLB shootdowns generally applicable and finding the correct one for a particular application no trivial matter.

Given the above, minimizing virtualizaton overhead induced by TLB shootdowns equates to finding a good balance between coarse memory allocations which minimize the amount of times the heap may have to be resized and lean

allocations which can be deallocated rapidly so that new allocations may recycle their memory from the free list without requiring additional heap expansion. Finding this balance must be done iteratively on a per-application basis.

On a positive note, §4.2.3 has made clear that only few multithreaded applications suffer significantly from excessive TLB shootdown overhead. As such, in most cases it suffices to keep the 'good balance' described above loosely in mind when laying out the application architecture. If performance testing afterwards should reveal TLB shootdown issues, these may be retroactively addressed by analyzing the application in search for the behaviors outlined in §8.2.3 and tweaking any code responsible for these behaviors iteratively until the issue is resolved.

8.3.4 Non-Uniform Memory Access Locality

Strictly speaking, little can be done about poor memory locality in a virtualized setting at the application source code level. After all, §4.2.2 has made clear that the cause of this problem is situated at the VM level, out of reach of the application source code. Even when an application achieves perfect memory locality in a native setting, in a virtualized environment the guest may unwittingly schedule a thread on a particular NUMA node while all of its data is located on another. Therefore, one may argue that application developers must rely on system administrators to make sure their applications perform optimally with regard to memory locality in a virtualized setting.

In spite of the above, application developers targeting virtualized platforms should not neglect memory locality. Namely, if the application itself exhibits good memory locality, it is likely that NUMA management algorithms integrated into many contemporary virtualized systems (see §5.4.2)—imperfect as they may be—will be able to detect and mitigate NUMA opacity issues, yielding good memory locality after all. When the application itself exhibits poor memory locality on the other hand, no amount of host level effort will be able to rectify the situation.

Concretely, 'not neglecting memory locality' means that data dependencies between concurrent threads should be minimized. This naturally implies the use of data parallelism. Additionally, as noted in §8.2.4, collocating data used by distinct threads on a single memory page should be avoided. This may be done by e.g. dividing input data into chunks to be processed by different threads along page boundaries or adding padding to smaller pieces of data so that they fill an entire memory page regardless.

8.4 NODedup

Following through on the precedent set in chapters 6 and 7, this chapter translates its scientific contribution into an industrially applicable solution which can both be used to make an impact in the real world and validate the theoretical propositions upon which it is based. Said solution is in this case a re-implementation of the *Dedup* benchmark from the PARSEC benchmark suite using the guidelines laid out in the previous section, aiming to reduce the virtualization overhead this benchmark induces dramatically. We named this alternative implementation of *Dedup* 'NODedup', which is short for 'No-Overhead Dedup'. The source code is freely available[1]. Moreover, appendix C provides all of the NODedup source files that deviate from the original *Dedup* benchmark.

PARSEC *Dedup* supports both data encoding and decoding using a variety of parallelization techniques and compression algorithms. Because supporting all of these features does not significantly strengthen the evidence for the efficacy of the guidelines proposed in §8.3 compared to supporting a thoughtfully selected subset thereof, NODedup only implements data encoding using pthreads and GZIP compression. Naturally however, the techniques presented in §8.3 are equally applicable to any other aspect of the original benchmark.

Section 4.2.3 has shown that the vast majority of overhead the *Dedup* benchmark incurs is related to blocking synchronization and memory management. Considering the architecture of this benchmark as described in §8.1, this is not surprising. Namely, its emphasis on task parallelism requires threads to synchronize each time a chunk transitions between pipeline stages. Moreover, *Dedup* by nature performs large amounts of consecutive, comparable memory allocations to create chunks which must pass through the entire pipeline before they can be deallocated; behavior listed in §8.2.3 as likely to induce excessive TLB shootdown overhead. Therefore, implementing NODedup equates to applying the guidelines listed in §8.3.1 and §8.3.3 to the original *Dedup* benchmark. The remainder of this section documents this process.

8.4.1 Blocking Synchronization

Careful analysis of the *Dedup* source code reveals that indeed the ringbuffers it employs between pipeline stages employ mutexes and condition variables in order to serialize access to the data they contain (see §8.2.1). These synchronization primitives are responsible for the vast majority of overhead related to blocking

[1] https://github.com/StijnSchildermans/dedup_without_overhead

synchronization observed in §4.2.3. Therefore, as prescribed in §8.3.1, NODedup does away with this pipeline altogether and replaces it with data parallelism. This removes almost all need for thread synchronization at a minor cost in scalability.

NODedup implements all pipeline stages of the original *Dedup* benchmark as functions which it applies in sequential order to the input data. The first of these functions—creating coarse chunks from the input file—is performed on the main thread, which immediately adds the newly created chunks to an ordered linked list which will be used to track the chunks throughout the remainder of the encoding process. After this sequential stage, the main thread creates a thread pool sized in accordance with the number of available CPUs. It then divides the chunk list into equally sized sublists, for each of which it submits a job to the thread pool consisting of the function representing the second pipeline stage applied to that sublist. The main thread then blocks until all of these jobs are finished, before repeating the job creation and blocking process for pipeline stages 3 and 4. In this way, all manipulation of central data structures—the thread pool and the chunk list—happens from the main thread only, mimimizing the need for thread synchronization. After these parallel stages, the main thread writes each of the sublists to the output file. Note that because this data parallel application architecture maintains chunk order, NODedup can skip the entire reordering stage of the original *Dedup* benchmark.

8.4.2 Memory Management

The original *Dedup* implementation reads input from disk in large blocks of 128 MB. It then refines these blocks into chunks, which are in fact pointers to a certain address within this large input block. Only when all chunks constituting such an input block have been processed, it is freed. This means that when later pipeline stages allocate memory, the allocator must often draw from the top of the heap rather than the free list. Because these later allocations are mostly related to temporary data structures and are therefore short-lived, this allocation pattern leads to the arena imbalance issue (see §7.3.1). NODedup addresses this by allocating each chunk individually in the fragmentation stage rather than using pointers to some address within a large preallocated buffer. These much smaller allocations can be freed more quickly and their memory can be recycled through the free list. The downside of this approach however is that the entire input data set must be copied.

Another improvement NODedup makes to the memory allocation behavior of *Dedup* pertains to the data compression stage. Namely, whenever the original implementation determines a chunk to be unique, it allocates a buffer to store

the compressed version of that chunk. These buffers form a significant portion of the 'allocations in later stages' causing the arena imbalance issue referred to above. NODedup eliminates most of these allocations by employing large memory buffers holding the compressed version of multiple chunks at once. Note that these buffers are unlikely to have a significant negative effect on memory efficiency because they only have a short lifespan, as compression of unique chunks is one of the last stages in the data deduplication algorithm as described in §8.1. Moreover, these buffers are likely to be allocated from the free list since thanks to the modifications described in the previous paragraph, input chunks are freed during the compression stage, allowing compression buffers created for subsequent chunks to recycle their memory.

Attentive readers may have noticed that the above paragraphs appear to be oxymoronic. Namely, the first paragraph advises to divide few large chunks into many smaller ones, while the second one advises merging many small allocations into a few large ones. However, when both paragraphs are combined and the subtle differences and interactions between the alterations they describe are taken into account, it becomes clear how two steps in opposite directions do not lead to the original starting point in this case. For example, while not explicitly stated above, compression buffers are still much smaller than the input buffers they replace and because they are created at a much later stage in the algorithm, they are less likely to force later allocations to be served from the top of the heap. This is a perfect illustration of the admonition from §8.3.3 regarding the complexity and iterative nature of addressing excessive TLB shootdown overhead through altering application behavior.

8.5 Evaluation

This chapter is no exception to the approach this dissertation follows with all of the ameliorations it proposes in the sense that it presents a thorough evaluation of the guidelines described in the previous section in order to provide evidence for their efficacy. However, because the work upon which this chapter is based preceded that presented in any of the previous chapters, the approach this evaluation takes deviates from the prescriptions provided in §3.3. This section lays out said approach below, after which it presents the conceptual effectiveness and eventual performance impact of the guidelines formulated in §8.3 on the *Dedup* bechmark in turn.

8.5.1 Method

System Settings

The system employed for evaluating NODedup is a NUMA server with 2 Skylake-era Intel Xeon CPUs, each with 8 physical cores without hyperthreading. Each memory bank is 16 GB in size. The VMM is KVM, running in Ubuntu 16.04. All contemporary performance optimizations were enabled.

This evaluation considers two VM configurations: one sporting 4 vCPUs on a single NUMA node and one sporting 14 vCPUs spread over two nodes. In both cases, the guest OS is Ubuntu 16.04. The larger VM is limited to 14 vCPUs to minimize resource contention between the VM and host background processes. Native equivalents of these system configurations are evaluated as well for reference.

Workloads

Naturally, the workloads of interest for this evaluation are NODedup and the original *Dedup* implementation from the PARSEC benchmark suite. The level of parallelism for both is always set equal to the number of CPUs available for the experiment in question.

Because NODedup implements only part of the functionality the original *Dedup* benchmark provides, both versions are executed outside of the regular PARSEC framework. Specifically, each encodes a predetermined 600 MB tarball consisting of a number of replicas of a set of pdf files.

Measurement

Analogously to previous chapters, this chapter quantifies the benefits of NODedup over the original *Dedup* benchmark by executing both in identical circumstances and presenting the former's performance normalized to that of the latter. Equally analogously to previous chapters, all results are averaged over 10 iterations to ensure their reliability (see §3.3.3).

Irrespective of potential performance gains, it is prudent to begin any performance evaluation by determining to what extent the technique being evaluated achieves its goals at a conceptual level. Because as noted in §4.2 the vast majority of virtualization overhead incurred by *Dedup* manifests itself in the form of VM exits, the number of these events is naturally a perfect fit for evaluating the conceptual effectiveness of the guidelines presented in §8.3.

Table 8.1: VM exits induced by NODedup relative to the original *Dedup*.

Event	4 vCPUs	14 vCPUs
VM_EXIT	-91%	-96%

Table 8.2: Execution time of NODedup relative to the original *Dedup* benchmark.

(v)CPUs	Native	Virtualized
4	+5%	-20%
14	-30%	-40%

Because—as stated multiple times throughout this dissertation—improvements at system level such as a reduction in VM exits do not necessarily translate to performance benefits visible to end users for multithreaded applications, it is important to evaluate the latter as well. The most fitting metric for this purpuse is—as equally stated multiple times before—application execution time, which is therefore also included in this evaluation.

8.5.2 Conceptual Effectiveness

Table 8.1 summarizes the number of VM exits NODedup induces relative to the original *Dedup* benchmark in both virtualized scenarios described above.

Table 8.1 indicates in no uncertain terms that NODedup suffers hardly any virtualization overhead compared to *Dedup*. Results improve even further as vCPU counts increase, which is not surprising since §4.2 has shown that both blocking synchronization and TLB shootdown overhead become more problematic as core counts increase. It is therefore clear that the guidelines presented in §8.3 can indeed be highly effective when applied correctly.

8.5.3 Performance

Table 8.5.3 shows the execution time of NODedup relative to the original *Dedup* benchmark in both the native and virtualized environments described above.

Table 8.5.3 reveals that the great reduction in VM exits NODedup yields as indicated by table 8.1 does not always impact execution time positively. Particularly, when natively run using 4 CPUs, a minor slowdown is observable. This is however to be expected, since in a native, single-socket environment VM exits are not relevant and TLB shootdown IPIs and blocking operations are

highly efficient. Therefore, the benefits NODedup yields by eliminating these operations are negligible, while NODedup sacrificed *Dedup*'s pipeline—which in itself conceptually improves performance—in return. In virtualized and NUMA environments on the other hand, table 8.5.3 paints a much different picture because there the impact of IPIs and VM exits is much greater, as discussed at length in previous chapters. This further validates the guidelines proposed in this chapter and stresses the importance of astute application design rather than—or complementary to—reliance on platform optimizations when it comes to minimizing virtualization overhead.

8.6 Related Work

As stated in previous chapters, optimizing virtualization technology is a popular topic in literature. However, all existing work focusses on solutions at the hardware or system software level. Chapter 5 elaborates on all of the promising examples of those proposed solutions. Because repeating all of these studies here adds no value to the dissertation as a whole, readers arriving here without having read chapter 5 are strongly encouraged to do so.

Besides listing all noteworthy work related to that presented in this chapter, chapter 5 repeatedly states what sets this chapter apart from any previously published study: regarding addressing virtualization overhead—let alone for multithreaded applications—purely at application source code level, no precedents exist in literature to the best knowledge of the author. While tools and frameworks exist that do reduce virtualization overhead (e.g. P3ARSEC [171]), they achieve this as an unintended side effect rather than a design goal. In fact, showing that P3ARSEC positively influences virtualization overhead for multithreaded applications is one of the contributions of chapter 5 that eventually led to the creation of this one. While it would be highly interesting to explore this avenue of related work in depth, doing so would require showing that these design patterns, frameworks and tools proposed in literature indeed have a positive effect on virtualization overhead for multithreaded applications in a manner similar to how §5.1.6 assessed P3ARSEC. This evidently is a scientific contribution on its own and goes beyond the scope of this section.

8.7 Conclusion

This chapter has shown that for computation-intensive multithreaded applications, certain design choices can have a dramatic effect on overhead and performance in a virtualized setting. Moreover, through NODedup this chapter has provided strong evidence that by adhering to a certain set of principles, applications are unlikely to suffer significant virtualization overhead.

Despite the positive results NODedup achieves, the mitigation technique this chapter provides remains somewhat vague in comparison the previous contributions presented in this dissertation. Unfortunately, this vagueness is largely inherent to the concept of guidelines, since every application is unique and it is up to practitioners to translate said guidelines into concrete virtualization-friendly application source code. Nevertheless, the author deems this chapter a valuable contribution to the field, not in the least because of its pragmatic nature and its potential for making an immediate and tangible impact in industry.

8.7.1 Personal Contribution

As stated in the introduction to this chapter, the guidelines presented here have gradually sprouted from the knowledge the author accumulated throughout this Ph. D. project. This evidently implies that this chapter entirely consists of original work of the main author.

8.7.2 Future Work

In the opinion of the author, one of the most interesting aspects of this chapter is the fact that it opens the door to a multitude of avenues for future work. Below a summary of the most interesting of these:

- While NODedup performs very well in its current form, further refinements are still possible. This could lead to interesting new insights regarding the guidelines presented in this chapter, especially those concerning data sharing;
- While NODedup provides a strong indication of the efficacy of the proposed guidelines, more similar experiments are desirable to further refine and validate them;

- Based on the description of source code likely to induce significant virtualization overhead in §8.2, a tool could be developed to analyze application source code in order to identify constructs that are likely to lead to significant virtualization overhead. Based on the guidelines presented in §8.3, this tool could even automatically improve this code, or otherise provide suggestions to developers on how to do so;

- The knowledge obtained from this chapter could be integrated into programming language primitives or libraries so that the abstractions they provide suffer less virtualization overhead. Alternatively, novel virtualization-friendly abstractions could be developed from the ground up. An excellent example of the former is ptlbmalloc2 (see chapter 7). The parallel patterns discussed in §5.1.6 are on the other hand an example of the latter, albeit without explicitly targeting minimizing virtualization overhead;

- As stated in §8.6, it is likely that design patterns, tools and frameworks with a positive effect on virtualization overhead for multithreaded applications have already been proposed in literature, albeit without their creators being aware of this because they never considered the implications of their contribution on the virtualization process. Identifying promising examples of such contributions and assessing them in a virtualized context is another promising avenue for future work.

Chapter 9

Conclusion

This dissertation has laid out a variety of scientific contributions traversing many aspects of contemporary virtualization technology. In conclusion to all of this work, it is prudent to reflect on whether or not this endeavor has been able to address the problems it set out to tackle, which equates to determining to what extent it has answered the research questions formulated in §1.3. To that end this chapter lists each of the partial research questions referred to above, followed by a discussion of how the work presented in this dissertation has addressed it. Naturally, cumulation of the answers to these partial research questions leads to the answer to the principal research question §1.3 describes, which is indeed the essence of this Ph. D. project. This final calculation is left up to the reader.

What causes high hardware-assisted virtualization cost for multi-threaded applications on the x86 platform?
Both chapter 3 and chapter 4 have been primarily concerned with addressing this first partial research question. The most important contribution of the former has been to clearly define virtualization overhead as a combination of system effects and application effects, expressed respectively as reduced resource efficiency and reduced temporal efficiency. The latter applied this knowledge in the form of a thorough analysis of the virtualization overhead suffered by multithreaded applications on modern platforms, which provided much needed insight into the state of the art regarding this topic. It affirmed many of the known causes of virtualization overhead for such applications and even identified several previously unknown ones. All of these can be broadly grouped into four categories: blocking synchronization, spinning synchronization, data sharing and NUMA locality. While the fact that chapter 4 can never guarantee that

it did not miss any relevant causes of overhead will always remain a threat to validity, the rigorous process it applied makes concluding that this first partial research question has been adequately answered for at least the vast majority of realistic workloads reasonable.

How effective are existing hardware-assisted x86 virtualization techniques at addressing the issues arising from virtualizing multithreaded applications?
This second partial research question was answered in great detail by both chapter 4 and chapter 5. The former did so by performing its performance analysis on a state-of-the-art platform including all of the existing techniques the question refers to, while the latter elaborated on several of these techniques at length. While we were pleased to find that great progress has been made in recent years, chapter 4 revealed that multithreaded applications still incur significant virtualization overhead, especially using larger VMs. Moreover, chapter 5 has made clear that many of the existing mitigation techniques are far from perfect; only being partially effective, being too restrictive in scope or having undesirable side effects.

Which techniques can reduce the cost of hardware-assisted virtualization of multithreaded applications on the x86 platform?
Chapter 5 has been entirely dedicated to answering this question. It proposed many promising research directions and suggested several ameliorations to existing technologies. While obviously this taxonomy of potential mitigation techniques can never be guaranteed to be exhaustive, it covers all of the causes of virtualization overhead chapter 4 identified. Therefore, it is in the estimation of the author fair to consider this research question adequately answered as well.

How can evidence for the efficacy of proposed techniques to reduce the cost of hardware-assisted virtualization of multithreaded applications on the x86 platform be provided?
Chapters 6, 7 and 8 are each dedicated to fleshing out one of the mitigation techniques chapter 5 proposed. Each of these chapters includes a thorough emperical performance evaluation comparing the technique it discusses to the state of the art, each time providing strong evidence in favor of the former: paratick and ptlbmalloc2 improve performance of multithreaded applications in a virtualized context by up to 15% and 45%, respectively. While much less generalizable, NODedup performs 40% better than the original *Dedup* benchmark upon which it is based, indicating the potential of the guidelines presented in chapter 8. As such, this final partial research question has certainly been adequately answered as well.

9.1 Valorization

At the faculty of Engineering Technology where this Ph. D. has taken place, industrial applicability is an important aspect of any research project. Concerning this, the most significant contributions of this dissertation are paratick (see §6.4), ptlbmalloc2 (see §7.5) and the guidelines to developers presented in §8.3. All of these mitigation techniques have been open sourced so that they can readily be adopted by practitioners and even be incorporated into existing projects. In the case of ptlbmalloc2, allowing for effortless adoption was even a core objective considerably influencing its design. Cloud providers and consumers alike are likely to benefit from adopting any of these techniques since the drastic reduction in virtualization overhead they effectuate for certain applications may yield them significant cost savings.

Naturally, the flip side of open source software is that it does not immediately allow for monitization to the benefit of its developer. As such, despite its notable contributions to the scientific and industrial landscapes, this Ph. D. project has not led to any marketable products or intellectual property. Notwithstanding, the knowledge and experience obtained while working on this project have undoubtedly opened the door to many future opportunities for the author.

9.2 Future Work

Software engineering is a rapidly evolving field of science. Indeed, the very concept of hardware virtualization—which is so central to this dissertation—is only about half a century old at the time of writing this work and has reinvented itself several times already. For example, only a few years ago it seemed likely that hardware virtualization would fade into obscurity due to the rapid surge of containerization, only to very recently charge back to the forefront of exciting developments in cloud computing thanks to unikernels. As uncertain as the future may be, hardware virtualization is likely to remain an indispensible aspect of software engineering for the foreseeable future, which means that addressing its issues will equally remain a relevant research topic for years to come.

This dissertation has provided numerous suggestions to future researchers. Rather than repeating these, this chapter refers to the 'Related Work' sections of previous chapters. Concerning how these suggestions relate to one another, the author deems pursuing those proposed in chapters 6, 7 and 8 most worthwhile, since the work presented in these respective chapters already provides a solid foundation for any such efforts. Of all of these, perhaps those proposed in chapter

8 are the most interesting, because that chapter approaches virtualization overhead from an entirely new angle compared to existing literature, which naturally comes with the biggest challenges but also the most potential. In any case, while hardware virtualization entirely free of overhead will likely prove a utopian idea, this dissertation has taken several more incremental steps in the right direction, providing copious incentives to future researchers to do the same along the way; as this dissertation itself was built upon the shoulders of a great number of works before it.

Appendix A

Paratick Source Code

This appendix discloses the paratick source code, structured in terms of the original Linux kernel 5.10.26 source files into which it is integrated.

A.1 Host

A.1.1 /include/linux/kvm_host.h

```
...

//Code above has not been altered.

struct kvm_vcpu {
  struct kvm *kvm;
#ifdef CONFIG_PREEMPT_NOTIFIERS
    struct preempt_notifier preempt_notifier;
#endif
    int cpu;
    int vcpu_id; /* id given by userspace at creation */
    int vcpu_idx; /* index in kvm->vcpus array */
    int srcu_idx;
    int mode;
    u64 requests;
    unsigned long guest_debug;

    int pre_pcpu;
    struct list_head blocked_vcpu_list;
```

```
20
21      struct mutex mutex;
22      struct kvm_run *run;
23
24      struct rcuwait wait;
25      struct pid __rcu *pid;
26      int sigset_active;
27      sigset_t sigset;
28      struct kvm_vcpu_stat stat;
29      unsigned int halt_poll_ns;
30      bool valid_wakeup;
31  #ifdef CONFIG_HAS_IOMEM
32      int mmio_needed;
33      int mmio_read_completed;
34      int mmio_is_write;
35      int mmio_cur_fragment;
36      int mmio_nr_fragments;
37      struct kvm_mmio_fragment mmio_fragments[
    KVM_MAX_MMIO_FRAGMENTS];
38  #endif
39  #ifdef CONFIG_KVM_ASYNC_PF
40      struct {
41        u32 queued;
42        struct list_head queue;
43        struct list_head done;
44        spinlock_t lock;
45      } async_pf;
46  #endif
47  #ifdef CONFIG_HAVE_KVM_CPU_RELAX_INTERCEPT
48      /*
49       * Cpu relax intercept or pause loop exit optimization
50       * in_spin_loop: set when a vcpu does a pause loop exit
51       *  or cpu relax intercepted.
52       * dy_eligible: indicates whether vcpu is eligible for
    directed yield.
53       */
54      struct {
55        bool in_spin_loop;
56        bool dy_eligible;
57      } spin_loop;
58  #endif
59    bool preempted;
60    bool ready;
61    struct kvm_vcpu_arch arch;
62    ktime_t last_tick;
63  };
```

```
64
65 //Code below has not been altered.
66
67 ...
```

A.1.2 /arch/x86/kvm/x86.c

```
1  ...
2
3  //Code above has not been altered.
4
5  static struct kvm_lapic_irq paratick_irq = {
6    .shorthand = APIC_DEST_SELF,
7    .dest_mode = APIC_DEST_PHYSICAL,
8    .delivery_mode = APIC_DM_FIXED,
9    .vector = 235,
10   .level = 15
11 };
12
13
14 static int vcpu_run(struct kvm_vcpu *vcpu)
15 {
16   int r;
17   ktime_t now;
18   struct kvm *kvm = vcpu->kvm;
19
20   vcpu->srcu_idx = srcu_read_lock(&kvm->srcu);
21   vcpu->arch.l1tf_flush_l1d = true;
22
23   for (;;) {
24     if (kvm_vcpu_running(vcpu)) {
25       r = vcpu_enter_guest(vcpu);
26     } else {
27       r = vcpu_block(kvm, vcpu);
28     }
29
30     if (r <= 0)
31       break;
32
33     kvm_clear_request(KVM_REQ_PENDING_TIMER, vcpu);
34
35     now = ktime_get();
36     if (kvm_cpu_has_pending_timer(vcpu))
37     {
38       vcpu->last_tick = now;
39       kvm_inject_pending_timer_irqs(vcpu);
```

```
40      }
41      else if (now - vcpu->last_tick > 4000000)
42      {
43        vcpu->last_tick = now;
44        kvm_apic_set_irq(vcpu, &paratick_irq, NULL);
45      }
46
47      if (dm_request_for_irq_injection(vcpu) &&
48        kvm_vcpu_ready_for_interrupt_injection(vcpu)) {
49        r = 0;
50        vcpu->run->exit_reason = KVM_EXIT_IRQ_WINDOW_OPEN;
51        ++vcpu->stat.request_irq_exits;
52        break;
53      }
54
55      if (__xfer_to_guest_mode_work_pending()) {
56        srcu_read_unlock(&kvm->srcu, vcpu->srcu_idx);
57        r = xfer_to_guest_mode_handle_work(vcpu);
58        if (r)
59                return r;
60        vcpu->srcu_idx = srcu_read_lock(&kvm->srcu);
61      }
62    }
63
64    srcu_read_unlock(&kvm->srcu, vcpu->srcu_idx);
65
66    return r;
67  }
68
69  //Code above has not been altered.
70
71  ...
```

A.2 Guest

A.2.1 /kernel/time/tick-sched.c

```
1  // SPDX-License-Identifier: GPL-2.0
2  /*
3   *  Copyright(C) 2005-2006, Thomas Gleixner <tglx@linutronix.
       de>
4   *  Copyright(C) 2005-2007, Red Hat, Inc., Ingo Molnar
5   *  Copyright(C) 2006-2007  Timesys Corp., Thomas Gleixner
6   */
7  #include <linux/cpu.h>
```

```
8  #include <linux/err.h>
9  #include <linux/hrtimer.h>
10 #include <linux/interrupt.h>
11 #include <linux/kernel_stat.h>
12 #include <linux/percpu.h>
13 #include <linux/nmi.h>
14 #include <linux/profile.h>
15 #include <linux/sched/signal.h>
16 #include <linux/sched/clock.h>
17 #include <linux/sched/stat.h>
18 #include <linux/sched/nohz.h>
19 #include <linux/module.h>
20 #include <linux/irq_work.h>
21 #include <linux/posix-timers.h>
22 #include <linux/context_tracking.h>
23 #include <linux/mm.h>
24 #include <linux/irq.h>
25 #include <linux/irqdesc.h>
26 #include <asm/irq_regs.h>
27 #include <asm/apic.h>
28 #include "tick-internal.h"
29 #include <trace/events/timer.h>
30
31 //Per-CPU nohz control structure
32 static DEFINE_PER_CPU(struct tick_sched, tick_cpu_sched);
33
34 struct tick_sched *tick_get_tick_sched(int cpu)
35 {
36   return &per_cpu(tick_cpu_sched, cpu);
37 }
38
39 #if defined(CONFIG_NO_HZ_COMMON) || defined(
      CONFIG_HIGH_RES_TIMERS)
40 //The time, when the last jiffy update happened. Protected by
       jiffies_lock.
41 static ktime_t last_jiffies_update;
42
43 //Must be called with interrupts disabled !
44 static void tick_do_update_jiffies64(ktime_t now)
45 {
46   unsigned long ticks = 0;
47   ktime_t delta;
48
49   //Do a quick check without holding jiffies_lock
50   delta = ktime_sub(now, READ_ONCE(last_jiffies_update));
51   if (delta < tick_period)
```

```
52      return;
53
54    /* Reevaluate with jiffies_lock held */
55    raw_spin_lock(&jiffies_lock);
56    write_seqcount_begin(&jiffies_seq);
57
58    delta = ktime_sub(now, last_jiffies_update);
59    if (delta >= tick_period) {
60
61      delta = ktime_sub(delta, tick_period);
62      /* Pairs with the lockless read in this function. */
63      WRITE_ONCE(last_jiffies_update,
64            ktime_add(last_jiffies_update, tick_period));
65
66      /* Slow path for long timeouts */
67      if (unlikely(delta >= tick_period)) {
68        s64 incr = ktime_to_ns(tick_period);
69
70        ticks = ktime_divns(delta, incr);
71
72        /* Pairs with the lockless read in this function. */
73        WRITE_ONCE(last_jiffies_update,
74              ktime_add_ns(last_jiffies_update,
75                incr * ticks));
76      }
77      do_timer(++ticks);
78
79      /* Keep the tick_next_period variable up to date */
80      tick_next_period = ktime_add(last_jiffies_update,
        tick_period);
81    } else {
82      write_seqcount_end(&jiffies_seq);
83      raw_spin_unlock(&jiffies_lock);
84      return;
85    }
86    write_seqcount_end(&jiffies_seq);
87    raw_spin_unlock(&jiffies_lock);
88    update_wall_time();
89  }
90
91  //Initialize and return retrieve the jiffies update.
92  static ktime_t tick_init_jiffy_update(void)
93  {
94    ktime_t period;
95
96    raw_spin_lock(&jiffies_lock);
```

```
97    write_seqcount_begin(&jiffies_seq);
98    /* Did we start the jiffies update yet ? */
99    if (last_jiffies_update == 0)
100     last_jiffies_update = tick_next_period;
101   period = last_jiffies_update;
102   write_seqcount_end(&jiffies_seq);
103   raw_spin_unlock(&jiffies_lock);
104   return period;
105 }
106
107 static void tick_sched_do_timer(struct tick_sched *ts,
        ktime_t now)
108 {
109   int cpu = smp_processor_id();
110
111 #ifdef CONFIG_NO_HZ_COMMON
112   //Check if the do_timer duty was dropped.
113   if (unlikely(tick_do_timer_cpu == TICK_DO_TIMER_NONE)) {
114     tick_do_timer_cpu = cpu;
115   }
116 #endif
117
118   /* Check, if the jiffies need an update */
119   if (tick_do_timer_cpu == cpu)
120     tick_do_update_jiffies64(now);
121
122   if (ts->inidle)
123     ts->got_idle_tick = 1;
124 }
125
126 static void tick_sched_handle(struct tick_sched *ts, struct
        pt_regs *regs)
127 {
128 #ifdef CONFIG_NO_HZ_COMMON
129   if (ts->tick_stopped) {
130     touch_softlockup_watchdog_sched();
131     if (is_idle_task(current))
132       ts->idle_jiffies++;
133     ts->next_tick = 0;
134   }
135 #endif
136   update_process_times(user_mode(regs));
137   profile_tick(CPU_PROFILING);
138 }
139 #endif
140
```

```
141 //NOHZ - aka dynamic tick functionality
142 #ifdef CONFIG_NO_HZ_COMMON
143 //NO HZ enabled ?
144 bool tick_nohz_enabled __read_mostly  = true;
145 unsigned long tick_nohz_active  __read_mostly;
146 //Enable / Disable tickless mode
147 static int __init setup_tick_nohz(char *str)
148 {
149   return (kstrtobool(str, &tick_nohz_enabled) == 0);
150 }
151
152 __setup("nohz=", setup_tick_nohz);
153
154 bool tick_nohz_tick_stopped(void)
155 {
156   struct tick_sched *ts = this_cpu_ptr(&tick_cpu_sched);
157   return ts->tick_stopped;
158 }
159
160 bool tick_nohz_tick_stopped_cpu(int cpu)
161 {
162   struct tick_sched *ts = per_cpu_ptr(&tick_cpu_sched, cpu);
163   return ts->tick_stopped;
164 }
165
166 //tick_nohz_update_jiffies - update jiffies when idle was
      interrupted
167 static void tick_nohz_update_jiffies(ktime_t now)
168 {
169   unsigned long flags;
170
171   __this_cpu_write(tick_cpu_sched.idle_waketime, now);
172
173   local_irq_save(flags);
174   tick_do_update_jiffies64(now);
175   local_irq_restore(flags);
176
177   touch_softlockup_watchdog_sched();
178 }
179
180 //Updates the per-CPU time idle statistics counters
181 static void update_ts_time_stats(int cpu, struct tick_sched *
      ts, ktime_t now, u64 *last_update_time)
182 {
183   ktime_t delta;
184
```

```
185   if (ts->idle_active) {
186     delta = ktime_sub(now, ts->idle_entrytime);
187     if (nr_iowait_cpu(cpu) > 0)
188       ts->iowait_sleeptime = ktime_add(ts->iowait_sleeptime,
      delta);
189     else
190       ts->idle_sleeptime = ktime_add(ts->idle_sleeptime,
      delta);
191     ts->idle_entrytime = now;
192   }
193
194   if (last_update_time)
195     *last_update_time = ktime_to_us(now);
196 }
197
198 static void tick_nohz_stop_idle(struct tick_sched *ts,
      ktime_t now)
199 {
200   update_ts_time_stats(smp_processor_id(), ts, now, NULL);
201   ts->idle_active = 0;
202
203   sched_clock_idle_wakeup_event();
204 }
205
206 static void tick_nohz_start_idle(struct tick_sched *ts)
207 {
208   ts->idle_entrytime = ktime_get();
209   ts->idle_active = 1;
210   sched_clock_idle_sleep_event();
211 }
212
213 u64 get_cpu_idle_time_us(int cpu, u64 *last_update_time)
214 {
215   struct tick_sched *ts = &per_cpu(tick_cpu_sched, cpu);
216   ktime_t now, idle;
217
218   if (!tick_nohz_active)
219     return -1;
220
221   now = ktime_get();
222   if (last_update_time) {
223     update_ts_time_stats(cpu, ts, now, last_update_time);
224     idle = ts->idle_sleeptime;
225   } else {
226     if (ts->idle_active && !nr_iowait_cpu(cpu)) {
227       ktime_t delta = ktime_sub(now, ts->idle_entrytime);
```

```
228
229         idle = ktime_add(ts->idle_sleeptime, delta);
230       } else {
231         idle = ts->idle_sleeptime;
232       }
233     }
234
235     return ktime_to_us(idle);
236
237 }
238 EXPORT_SYMBOL_GPL(get_cpu_idle_time_us);
239
240 u64 get_cpu_iowait_time_us(int cpu, u64 *last_update_time)
241 {
242     struct tick_sched *ts = &per_cpu(tick_cpu_sched, cpu);
243     ktime_t now, iowait;
244
245     if (!tick_nohz_active)
246       return -1;
247
248     now = ktime_get();
249     if (last_update_time) {
250       update_ts_time_stats(cpu, ts, now, last_update_time);
251       iowait = ts->iowait_sleeptime;
252     } else {
253       if (ts->idle_active && nr_iowait_cpu(cpu) > 0) {
254         ktime_t delta = ktime_sub(now, ts->idle_entrytime);
255
256         iowait = ktime_add(ts->iowait_sleeptime, delta);
257       } else {
258         iowait = ts->iowait_sleeptime;
259       }
260     }
261
262     return ktime_to_us(iowait);
263 }
264 EXPORT_SYMBOL_GPL(get_cpu_iowait_time_us);
265
266 static void tick_nohz_restart(struct tick_sched *ts, ktime_t
      now)
267 {
268     ts->next_tick = 0;
269 }
270
271 static inline bool local_timer_softirq_pending(void)
272 {
```

```
273   return local_softirq_pending() & BIT(TIMER_SOFTIRQ);
274 }
275
276 static ktime_t tick_nohz_next_event(struct tick_sched *ts,
      int cpu)
277 {
278   u64 basemono, next_tick, next_tmr, next_rcu, delta, expires
      ;
279   unsigned long basejiff;
280   unsigned int seq;
281
282   /* Read jiffies and the time when jiffies were updated last
       */
283   do {
284     seq = read_seqcount_begin(&jiffies_seq);
285     basemono = last_jiffies_update;
286     basejiff = jiffies;
287   } while (read_seqcount_retry(&jiffies_seq, seq));
288   ts->last_jiffies = basejiff;
289   ts->timer_expires_base = basemono;
290
291   //Keep the periodic tick, when RCU, architecture or
      irq_work requests it.
292   if (rcu_needs_cpu(basemono, &next_rcu) || arch_needs_cpu()
      ||
293     irq_work_needs_cpu() || local_timer_softirq_pending()) {
294   next_tick = basemono + TICK_NSEC;
295   } else {
296     //Get the next pending timer.
297     next_tmr = get_next_timer_interrupt(basejiff, basemono);
298     ts->next_timer = next_tmr;
299     /* Take the next rcu event into account */
300     next_tick = next_rcu < next_tmr ? next_rcu : next_tmr;
301   }
302
303   /*
304    * If the tick is due in the next period, keep it ticking
      or
305    * force prod the timer.
306    */
307   delta = next_tick - basemono;
308   if (delta <= (u64)TICK_NSEC) {
309     /*
310      * Tell the timer code that the base is not idle, i.e.
      undo
311      * the effect of get_next_timer_interrupt():
```

```
312         */
313         timer_clear_idle();
314         /*
315          * We've not stopped the tick yet, and there's a timer in
             the
316          * next period, so no point in stopping it either, bail.
317          */
318         if (!ts->tick_stopped) {
319           ts->timer_expires = 0;
320           goto out;
321         }
322       }
323
324       /*
325        * If this CPU is the one which had the do_timer() duty
           last, we limit
326        * the sleep time to the timekeeping max_deferment value.
327        * Otherwise we can sleep as long as we want.
328        */
329       delta = timekeeping_max_deferment();
330       if (cpu != tick_do_timer_cpu &&
331           (tick_do_timer_cpu != TICK_DO_TIMER_NONE || !ts->
             do_timer_last))
332         delta = KTIME_MAX;
333
334       /* Calculate the next expiry time */
335       if (delta < (KTIME_MAX - basemono))
336         expires = basemono + delta;
337       else
338         expires = KTIME_MAX;
339
340       ts->timer_expires = min_t(u64, expires, next_tick);
341     out:
342       return ts->timer_expires;
343     }
344
345     static void tick_nohz_stop_tick(struct tick_sched *ts, int
           cpu)
346     {
347       struct clock_event_device *dev = __this_cpu_read(
           tick_cpu_device.evtdev);
348       u64 basemono = ts->timer_expires_base;
349       u64 expires = ts->timer_expires;
350       ktime_t tick = expires;
351
```

```
352   /* Make sure we won't be trying to stop it twice in a row.
        */
353   ts->timer_expires_base = 0;
354
355   if (cpu == tick_do_timer_cpu) {
356     tick_do_timer_cpu = TICK_DO_TIMER_NONE;
357     ts->do_timer_last = 1;
358   } else if (tick_do_timer_cpu != TICK_DO_TIMER_NONE)
359     ts->do_timer_last = 0;
360
361   /* Skip reprogram of event if its not changed */
362   if (ts->tick_stopped && (expires == ts->next_tick)) {
363     /* Sanity check: make sure clockevent is actually
        programmed */
364     if (tick == KTIME_MAX || ts->next_tick ==
        hrtimer_get_expires(&ts->sched_timer))
365       return;
366
367     WARN_ON_ONCE(1);
368     printk_once("basemono: %llu ts->next_tick: %llu dev->
        next_event: %llu timer->active: %d timer->expires: %llu\n
        ",
369           basemono, ts->next_tick, dev->next_event,
370           hrtimer_active(&ts->sched_timer),
        hrtimer_get_expires(&ts->sched_timer));
371   }
372
373   if (!ts->tick_stopped) {
374     calc_load_nohz_start();
375     quiet_vmstat();
376
377     ts->last_tick = hrtimer_get_expires(&ts->sched_timer);
378     ts->tick_stopped = 1;
379     trace_tick_stop(1, TICK_DEP_MASK_NONE);
380   }
381
382   ts->next_tick = tick;
383
384   /*
385    * If the expiration time == KTIME_MAX, then we simply stop
386    * the tick timer.
387    */
388   if (unlikely(expires == KTIME_MAX)) {
389     return;
390   }
391
```

```
392   if (ts->nohz_mode == NOHZ_MODE_HIGHRES
393       && (!hrtimer_active(&ts->sched_timer)
394     || hrtimer_get_expires(&ts->sched_timer) >= tick))
395   {
396     hrtimer_start(&ts->sched_timer, tick,
397             HRTIMER_MODE_ABS_PINNED_HARD);
398   } else {
399     hrtimer_set_expires(&ts->sched_timer, tick);
400     tick_program_event(tick, 1);
401   }
402 }
403
404 static void tick_nohz_retain_tick(struct tick_sched *ts)
405 {
406   ktime_t now, next_event;
407   now = ktime_get();
408
409   if (!hrtimer_active(&ts->sched_timer) ||
      hrtimer_get_expires(&ts->sched_timer) > now + tick_period
      )
410   {
411     next_event = tick_nohz_next_event(ts, smp_processor_id())
      ;
412     if (next_event == 0)
413       hrtimer_start(&ts->sched_timer, now + tick_period,
      HRTIMER_MODE_ABS_PINNED_HARD);
414     else if (next_event < KTIME_MAX)
415       hrtimer_start(&ts->sched_timer, next_event,
      HRTIMER_MODE_ABS_PINNED_HARD);
416   }
417   ts->timer_expires_base = 0;
418 }
419
420 static void tick_nohz_restart_sched_tick(struct tick_sched *
      ts, ktime_t now)
421 {
422   /* Update jiffies first */
423   tick_do_update_jiffies64(now);
424
425   /*
426    * Clear the timer idle flag, so we avoid IPIs on remote
      queueing and
427    * the clock forward checks in the enqueue path:
428    */
429   timer_clear_idle();
430
```

```
  calc_load_nohz_stop();
  touch_softlockup_watchdog_sched();
  /*
   * Cancel the scheduled timer and restore the tick
   */
  ts->tick_stopped  = 0;
  ts->idle_exittime = now;

  tick_nohz_restart(ts, now);
}

static bool can_stop_idle_tick(int cpu, struct tick_sched *ts
    )
{
  if (unlikely(!cpu_online(cpu))) {
    if (cpu == tick_do_timer_cpu)
      tick_do_timer_cpu = TICK_DO_TIMER_NONE;
    /*
     * Make sure the CPU doesn't get fooled by obsolete tick
     * deadline if it comes back online later.
     */
    ts->next_tick = 0;
    return false;
  }

  if (unlikely(ts->nohz_mode == NOHZ_MODE_INACTIVE))
    return false;

  if (need_resched())
    return false;

  if (unlikely(local_softirq_pending())) {
    static int ratelimit;

    if (ratelimit < 10 &&
        (local_softirq_pending() & SOFTIRQ_STOP_IDLE_MASK)) {
      pr_warn("NOHZ tick-stop error: Non-RCU local softirq
    work is pending, handler #%02x!!!\n",
        (unsigned int) local_softirq_pending());
      ratelimit++;
    }
    return false;
  }

  if (tick_nohz_full_enabled()) {
    /*
```

```
475         * Keep the tick alive to guarantee timekeeping
        progression
476         * if there are full dynticks CPUs around
477         */
478        if (tick_do_timer_cpu == cpu)
479          return false;
480
481        /* Should not happen for nohz-full */
482        if (WARN_ON_ONCE(tick_do_timer_cpu == TICK_DO_TIMER_NONE)
        )
483          return false;
484      }
485
486      return true;
487    }
488
489    static void __tick_nohz_idle_stop_tick(struct tick_sched *ts)
490    {
491      ktime_t expires;
492      int cpu = smp_processor_id();
493
494      /*
495       * If tick_nohz_get_sleep_length() ran tick_nohz_next_event
        (), the
496       * tick timer expiration time is known already.
497       */
498      if (ts->timer_expires_base)
499        expires = ts->timer_expires;
500      else if (can_stop_idle_tick(cpu, ts))
501        expires = tick_nohz_next_event(ts, cpu);
502      else
503        return;
504
505      ts->idle_calls++;
506
507      if (expires > 0LL) {
508        int was_stopped = ts->tick_stopped;
509
510        tick_nohz_stop_tick(ts, cpu);
511
512        ts->idle_sleeps++;
513        ts->idle_expires = expires;
514
515        if (!was_stopped && ts->tick_stopped) {
516          ts->idle_jiffies = ts->last_jiffies;
517          nohz_balance_enter_idle(cpu);
```

```
518       }
519     } else {
520       tick_nohz_retain_tick(ts);
521     }
522   }
523
524   void tick_nohz_idle_stop_tick(void)
525   {
526     __tick_nohz_idle_stop_tick(this_cpu_ptr(&tick_cpu_sched));
527   }
528
529   void tick_nohz_idle_retain_tick(void)
530   {
531     tick_nohz_retain_tick(this_cpu_ptr(&tick_cpu_sched));
532     /*
533      * Undo the effect of get_next_timer_interrupt() called
        from
534      * tick_nohz_next_event().
535      */
536     timer_clear_idle();
537   }
538
539   //Prepare for entering idle on the current CPU
540   void tick_nohz_idle_enter(void)
541   {
542     struct tick_sched *ts;
543     lockdep_assert_irqs_enabled();
544     local_irq_disable();
545     ts = this_cpu_ptr(&tick_cpu_sched);
546     WARN_ON_ONCE(ts->timer_expires_base);
547     ts->inidle = 1;
548     tick_nohz_start_idle(ts);
549     local_irq_enable();
550   }
551
552   //Update next tick event from interrupt exit
553   void tick_nohz_irq_exit(void)
554   {
555     struct tick_sched *ts = this_cpu_ptr(&tick_cpu_sched);
556
557     if (ts->inidle)
558       tick_nohz_start_idle(ts);
559     else
560       tick_nohz_full_update_tick(ts);
561   }
562
```

```
563 //Check whether or not the tick handler has run
564 bool tick_nohz_idle_got_tick(void)
565 {
566   struct tick_sched *ts = this_cpu_ptr(&tick_cpu_sched);
567
568   if (ts->got_idle_tick) {
569     ts->got_idle_tick = 0;
570     return true;
571   }
572   return false;
573 }
574
575 //Return the next expiration time for the hrtimer or the tick
      , whatever that expires first.
576 ktime_t tick_nohz_get_next_hrtimer(void)
577 {
578   return __this_cpu_read(tick_cpu_device.evtdev)->next_event;
579 }
580
581 //Return the expected length of the current sleep
582 ktime_t tick_nohz_get_sleep_length(ktime_t *delta_next)
583 {
584   struct clock_event_device *dev = __this_cpu_read(
       tick_cpu_device.evtdev);
585   struct tick_sched *ts = this_cpu_ptr(&tick_cpu_sched);
586   int cpu = smp_processor_id();
587   /*
588    * The idle entry time is expected to be a sufficient
       approximation of
589    * the current time at this point.
590    */
591   ktime_t now = ts->idle_entrytime;
592   ktime_t next_event;
593
594  WARN_ON_ONCE(!ts->inidle);
595
596   *delta_next = ktime_sub(dev->next_event, now);
597
598   if (!can_stop_idle_tick(cpu, ts))
599     return *delta_next;
600
601   next_event = tick_nohz_next_event(ts, cpu);
602   if (!next_event)
603     return *delta_next;
604
605   /*
```

```
606     * If the next highres timer to expire is earlier than
        next_event, the
607     * idle governor needs to know that.
608     */
609    next_event = min_t(u64, next_event,
610           hrtimer_next_event_without(&ts->sched_timer));
611
612    return ktime_sub(next_event, now);
613  }
614
615  unsigned long tick_nohz_get_idle_calls_cpu(int cpu)
616  {
617    struct tick_sched *ts = tick_get_tick_sched(cpu);
618    return ts->idle_calls;
619  }
620
621  unsigned long tick_nohz_get_idle_calls(void)
622  {
623    struct tick_sched *ts = this_cpu_ptr(&tick_cpu_sched);
624    return ts->idle_calls;
625  }
626
627  static void tick_nohz_account_idle_ticks(struct tick_sched *
       ts)
628  {
629  #ifndef CONFIG_VIRT_CPU_ACCOUNTING_NATIVE
630    unsigned long ticks;
631
632    if (vtime_accounting_enabled_this_cpu())
633      return;
634    /*
635     * We stopped the tick in idle. Update process times would
        miss the
636     * time we slept as update_process_times does only a 1 tick
637     * accounting. Enforce that this is accounted to idle !
638     */
639    ticks = jiffies - ts->idle_jiffies;
640    //We might be one off. Do not randomly account a huge
       number of ticks!
641    if (ticks && ticks < LONG_MAX)
642      account_idle_ticks(ticks);
643  #endif
644  }
645
646  static void __tick_nohz_idle_restart_tick(struct tick_sched *
       ts, ktime_t now)
```

```
647 {
648   tick_nohz_restart_sched_tick(ts, now);
649   tick_nohz_account_idle_ticks(ts);
650 }
651
652 void tick_nohz_idle_restart_tick(void)
653 {
654   struct tick_sched *ts = this_cpu_ptr(&tick_cpu_sched);
655
656   if (ts->tick_stopped)
657     __tick_nohz_idle_restart_tick(ts, ktime_get());
658 }
659
660 void tick_nohz_idle_exit(void)
661 {
662   struct tick_sched *ts = this_cpu_ptr(&tick_cpu_sched);
663   bool idle_active, tick_stopped;
664   ktime_t now;
665
666   local_irq_disable();
667
668   WARN_ON_ONCE(!ts->inidle);
669   WARN_ON_ONCE(ts->timer_expires_base);
670
671   ts->inidle = 0;
672   idle_active = ts->idle_active;
673   tick_stopped = ts->tick_stopped;
674
675   if (idle_active || tick_stopped)
676     now = ktime_get();
677
678   if (idle_active)
679     tick_nohz_stop_idle(ts, now);
680
681   if (tick_stopped)
682     __tick_nohz_idle_restart_tick(ts, now);
683
684   local_irq_enable();
685 }
686
687 //The nohz low res interrupt handler
688 static void tick_nohz_handler(struct clock_event_device *dev)
689 {
690   struct tick_sched *ts = this_cpu_ptr(&tick_cpu_sched);
691   struct pt_regs *regs = get_irq_regs();
692   ktime_t now = ktime_get();
```

```
693
694     dev->next_event = KTIME_MAX;
695
696     tick_sched_do_timer(ts, now);
697     tick_sched_handle(ts, regs);
698
699     /* No need to reprogram if we are running tickless  */
700     if (unlikely(ts->tick_stopped))
701       return;
702
703     hrtimer_forward(&ts->sched_timer, now, tick_period);
704     tick_program_event(hrtimer_get_expires(&ts->sched_timer),
        1);
705 }
706
707 static inline void tick_nohz_activate(struct tick_sched *ts,
      int mode)
708 {
709     if (!tick_nohz_enabled)
710       return;
711     ts->nohz_mode = mode;
712     /* One update is enough */
713     if (!test_and_set_bit(0, &tick_nohz_active))
714       timers_update_nohz();
715 }
716
717 //tick_nohz_switch_to_nohz - switch to nohz mode
718 static void tick_nohz_switch_to_nohz(void)
719 {
720     struct tick_sched *ts = this_cpu_ptr(&tick_cpu_sched);
721     ktime_t next;
722
723     if (!tick_nohz_enabled)
724       return;
725
726     if (tick_switch_to_oneshot(tick_nohz_handler))
727       return;
728
729     /*
730      * Recycle the hrtimer in ts, so we can share the
731      * hrtimer_forward with the highres code.
732      */
733     hrtimer_init(&ts->sched_timer, CLOCK_MONOTONIC,
        HRTIMER_MODE_ABS_HARD);
734     /* Get the next period */
735     next = tick_init_jiffy_update();
```

```
736
737   hrtimer_set_expires(&ts->sched_timer, next);
738   hrtimer_forward_now(&ts->sched_timer, tick_period);
739   tick_program_event(hrtimer_get_expires(&ts->sched_timer),
        1);
740   tick_nohz_activate(ts, NOHZ_MODE_LOWRES);
741 }
742
743 static inline void tick_nohz_irq_enter(void)
744 {
745   struct tick_sched *ts = this_cpu_ptr(&tick_cpu_sched);
746   ktime_t now;
747
748   if (!ts->idle_active && !ts->tick_stopped)
749     return;
750   now = ktime_get();
751   if (ts->idle_active)
752     tick_nohz_stop_idle(ts, now);
753   if (ts->tick_stopped)
754     tick_nohz_update_jiffies(now);
755 }
756
757 #else
758
759 static inline void tick_nohz_switch_to_nohz(void) { }
760 static inline void tick_nohz_irq_enter(void) { }
761 static inline void tick_nohz_activate(struct tick_sched *ts,
      int mode) { }
762
763 #endif /* CONFIG_NO_HZ_COMMON */
764
765 //Called from irq_enter to notify about the possible
      interruption of idle()
766 void tick_irq_enter(void)
767 {
768   tick_check_oneshot_broadcast_this_cpu();
769   tick_nohz_irq_enter();
770 }
771
772 //High resolution timer specific code
773 #ifdef CONFIG_HIGH_RES_TIMERS
774
775 static void do_tick(void)
776 {
777   struct tick_sched* ts = this_cpu_ptr(&tick_cpu_sched);
778   struct pt_regs* regs = get_irq_regs();
```

```
779   ktime_t now = ktime_get();
780
781   tick_sched_do_timer(ts, now);
782
783   if (regs)
784                 tick_sched_handle(ts, regs);
785         else
786                 ts->next_tick = 0;
787 }
788
789 /*
790  * We rearm the timer until we get disabled by the idle code.
791  * Called with interrupts disabled.
792  */
793 static enum hrtimer_restart tick_sched_timer(struct hrtimer *
        timer)
794 {
795   struct tick_sched* ts = this_cpu_ptr(&tick_cpu_sched);
796
797   if (ts->inidle)
798     do_tick();
799
800   return HRTIMER_NORESTART;
801 }
802
803 static int sched_skew_tick;
804
805 static int __init skew_tick(char *str)
806 {
807   get_option(&str, &sched_skew_tick);
808
809   return 0;
810 }
811 early_param("skew_tick", skew_tick);
812
813 void handle_paratick_irq(struct irq_desc* desc)
814 {
815   do_tick();
816   ack_APIC_irq();
817 }
818
819 static struct irq_desc paratick_desc = {
820   .handle_irq = handle_paratick_irq
821 };
822
823 static void install_paratick_handler(void)
```

```
824 {
825   struct irq_desc* (*descs)[256] = this_cpu_ptr(&vector_irq);
826   (*descs)[235] = &paratick_desc;
827 }
828
829 //tick_setup_sched_timer - setup the tick emulation timer
830 void tick_setup_sched_timer(void)
831 {
832   struct tick_sched *ts = this_cpu_ptr(&tick_cpu_sched);
833   ktime_t now = ktime_get();
834
835   // Emulate tick processing via per-CPU hrtimers:
836   hrtimer_init(&ts->sched_timer, CLOCK_MONOTONIC,
        HRTIMER_MODE_ABS_HARD);
837   ts->sched_timer.function = tick_sched_timer;
838
839   /* Get the next period (per-CPU) */
840   hrtimer_set_expires(&ts->sched_timer,
        tick_init_jiffy_update());
841
842   /* Offset the tick to avert jiffies_lock contention. */
843   if (sched_skew_tick) {
844     u64 offset = ktime_to_ns(tick_period) >> 1;
845     do_div(offset, num_possible_cpus());
846     offset *= smp_processor_id();
847     hrtimer_add_expires_ns(&ts->sched_timer, offset);
848   }
849
850   hrtimer_forward(&ts->sched_timer, now, tick_period);
851   hrtimer_start_expires(&ts->sched_timer,
        HRTIMER_MODE_ABS_PINNED_HARD);
852   tick_nohz_activate(ts, NOHZ_MODE_HIGHRES);
853
854   install_paratick_handler();
855 }
856 #endif /* HIGH_RES_TIMERS */
857
858 #if defined CONFIG_NO_HZ_COMMON || defined
        CONFIG_HIGH_RES_TIMERS
859 void tick_cancel_sched_timer(int cpu)
860 {
861   struct tick_sched *ts = &per_cpu(tick_cpu_sched, cpu);
862
863 # ifdef CONFIG_HIGH_RES_TIMERS
864   if (ts->sched_timer.base)
865     hrtimer_cancel(&ts->sched_timer);
```

```
# endif

  memset(ts, 0, sizeof(*ts));
}
#endif

//Async notification about clocksource changes
void tick_clock_notify(void)
{
  int cpu;
  for_each_possible_cpu(cpu)
    set_bit(0, &per_cpu(tick_cpu_sched, cpu).check_clocks);
}

//Async notification about clock event changes
void tick_oneshot_notify(void)
{
  struct tick_sched *ts = this_cpu_ptr(&tick_cpu_sched);
  set_bit(0, &ts->check_clocks);
}

int tick_check_oneshot_change(int allow_nohz)
{
  struct tick_sched *ts = this_cpu_ptr(&tick_cpu_sched);

  if (!test_and_clear_bit(0, &ts->check_clocks))
    return 0;

  if (ts->nohz_mode != NOHZ_MODE_INACTIVE)
    return 0;

  if (!timekeeping_valid_for_hres() || !
    tick_is_oneshot_available())
    return 0;

  if (!allow_nohz)
    return 1;

  tick_nohz_switch_to_nohz();
  return 0;
}
```

Appendix B

Ptlbmalloc2 Source Code

Below the entire ptlbmalloc2 code base. This code may be compiled to a static library and linked into any application based on glibc.

B.1 Headers

B.1.1 Global.h

```
1  #ifndef GLOBAL_H
2  #define GLOBAL_H
3
4  //GLOBAL VARIABLES
5  extern size_t TOP_PAD;
6  extern size_t HEAP_M_SIZE;
7  extern size_t MMAP_THRESHOLD;
8  extern size_t MAX_MMAP_THRESHOLD;
9  extern size_t TRIM_THRESHOLD;
10
11 //EXTERNAL FUNCTIONS
12 extern void* __libc_malloc(size_t size);
13 extern void __libc_free(void* ptr);
14 extern void* __libc_calloc(size_t num, size_t size);
15 extern void* __libc_realloc(void* ptr, size_t size);
16
17 #endif
```

B.1.2 Types.h

```
#ifndef TYPES_H
#define TYPES_H

typedef void* ptmalloc2_ptr;
typedef void* mchunk_ptr;
typedef size_t size_field;
typedef char flags_t;

//Placeholder for the glibc malloc_state struct
typedef struct _malloc_state_proxy
{
  int lock;
  int flags;
  int have_fastchunks;
  void* fastbins[10];
  void* top;
  void* last_remainder;
  void* bins[254];
  unsigned int binmap[4];
  struct _malloc_state_proxy *next;
  struct _malloc_state_proxy *next_free;
  size_t attached_threads;
  size_t system_mem;
  size_t max_system_mem;
} arena;

typedef struct _mem_state{
  size_t used;
  size_t top;
} mem_state;

//Placeholder for the glibc heap_info struct
typedef struct _heap_info_proxy
{
  arena* arena;
  struct _heap_info_proxy *prev;
  size_t size;
  size_t mprotect_size;
} heap_info_proxy;

#endif
```

B.1.3 CPU_monitor.h

```c
#ifndef CPU_MONITOR_H
#define CPU_MONITOR_H

extern unsigned short used_cpus;
extern unsigned short max_cpus;

void init_cpu_monitor();

#endif
```

B.1.4 Chunk.h

```c
#ifndef CHUNK_H
#define CHUNK_H

#include <stdbool.h>
#include "types.h"

#define MCHUNK_PTR_TO_PTMALLOC2_PTR(ptr) (ptr + 2 * sizeof(
    size_t))
#define IS_MMAPPED(chunk) (*((size_t*)chunk - 1) & 2)
#define HEAP_INFO(ptr) ((heap_info_proxy*)((long)ptr & ~(
    HEAP_M_SIZE -1)))
#define ARENA(ptr) (HEAP_INFO(ptr)->arena)
#define MAIN(ptr) (!(*((size_t*)ptr - 1 ) & 4))
#define PREV_INUSE(chunk) (*((size_t*)chunk - 1) & 1)
#define SIZE_FIELD(ptr) (*((size_t*)ptr -1))
#define SIZE(ptr) (SIZE_FIELD(ptr) & ~(7))
#define FLAGS(ptr) (SIZE_FIELD(ptr) & 7)
#define TOP(ar) (SIZE(MCHUNK_PTR_TO_PTMALLOC2_PTR(ar->top)))

void set_chunk_size(ptmalloc2_ptr ptr, size_t size);
void set_chunk_size_head(ptmalloc2_ptr ptr, size_t size);

#endif
```

B.1.5 Arena.h

```c
#ifndef ARENA_H
#define ARENA_H

#include <stddef.h>
#include <stdbool.h>
#include "types.h"

```

```
8  extern arena* main_arena;
9
10 void init_arenas(ptmalloc2_ptr ptr);
11 void add_arena(arena* ar);
12
13 bool arena_exists(arena* ar);
14 int num_arenas();
15
16 mem_state get_mem_state();
17 void trim_arenas();
18 void expand_arena(arena* ar);
19 bool need_trim();
20
21 #endif
```

B.1.6 Ptlbmalloc2.h

```
1  #ifndef PTLBMALLOC2_H
2  #define PTLBMALLOC2_H
3
4  extern void* malloc(size_t size);
5  extern void free(void* ptr);
6  extern void* calloc(size_t num, size_t size);
7  extern void* realloc(void* ptr, size_t size);
8
9  int set_sensitivity(float val);
10
11 #endif
```

B.2 Implementation

B.2.1 CPU_monitor.c

```
1  #include <sys/time.h>
2  #include <signal.h>
3  #include <stdio.h>
4  #include <stdbool.h>
5  #include <sys/times.h>
6  #include <sys/sysinfo.h>
7  #include <unistd.h>
8
9  unsigned short max_cpus;
10 float ticks_per_us;
11 struct tms last_times;
```

```
12 unsigned short used_cpus;
13
14 //Estimate the number of CPUs currently being used by the
       program.
15 static void calc_cpus(int sig){
16   //When the number of used CPUs can not be determined,
       assume all system CPUs are used.
17   used_cpus = max_cpus;
18
19   int passed_usecs = 1000000;
20   unsigned short cpus_used;
21
22   //Get CPU time passed
23   struct tms cur_times;
24   times(&cur_times);
25   float cpu_time = (cur_times.tms_utime + cur_times.tms_stime
       - last_times.tms_utime - last_times.tms_stime)/
     ticks_per_us;
26   cpus_used = cpu_time/passed_usecs;
27   if  (cpus_used > max_cpus || cpus_used == 0) return;
28
29   //If successful, set new values
30   last_times = cur_times;
31   used_cpus = cpus_used;
32 }
33
34 void init_cpu_monitor(){
35     times(&last_times);
36     max_cpus = get_nprocs();
37     ticks_per_us = sysconf(_SC_CLK_TCK)/1000000.0;
38     used_cpus = max_cpus;
39
40     signal(SIGALRM, calc_cpus);
41     struct itimerval timer;
42     timer.it_interval.tv_sec = 1;
43     timer.it_interval.tv_usec = 0;
44     timer.it_value.tv_sec = 1;
45     timer.it_value.tv_usec = 0;
46     setitimer(ITIMER_REAL, &timer, NULL);
47 }
```

B.2.2 Chunk.c

```
1 #include <stdio.h>
2 #include <stdbool.h>
3 #include "global.h"
```

```
4  #include "types.h"
5
6
7  void set_chunk_size(ptmalloc2_ptr ptr, size_t size){
8    *((size_t*)ptr - 1) = size;
9    *((size_t*)(ptr + size - 2)) = size;
10 }
11
12 void set_chunk_size_head(ptmalloc2_ptr ptr, size_t size){
13   *((size_t*)ptr - 1) = size;
14 }
```

B.2.3 Arena.c

```
1  #include <sys/mman.h>
2  #include <unistd.h>
3  #include <sys/syscall.h>
4  #include <linux/futex.h>
5  #include <sys/time.h>
6  #include <stdio.h>
7  #include <stdbool.h>
8  #include <malloc.h>
9  #include <pthread.h>
10 #include "cpu_monitor.h"
11 #include "chunk.h"
12 #include "global.h"
13
14
15 int max_arenas;
16 arena** arenas = NULL;
17 arena* main_arena;
18
19 static inline void set_main_arena(ptmalloc2_ptr ptr)
20 {
21      arena* a = HEAP_INFO(ptr)->arena;
22      arena* ar = a->next;
23      arena* max = a;
24      while (ar != a)
25      {
26   if ( ar > max) max = ar;
27   ar = ar->next;
28      }
29      main_arena = max;
30 }
31
32 static void* find_main_arena(void* arg)
```

```
33 {
34       ptmalloc2_ptr ptr = __libc_malloc(1024);
35       set_main_arena(ptr);
36       __libc_free(ptr);
37 }
38
39 void init_arenas(ptmalloc2_ptr ptr)
40 {
41       max_arenas = 8 * max_cpus;
42       arenas = mmap(NULL, max_arenas * sizeof(arena*),
     PROT_READ | PROT_WRITE, MAP_PRIVATE | MAP_ANONYMOUS, -1,
     0);
43
44       if (MAIN(ptr))
45       {
46         pthread_t thread;
47         pthread_create(&thread, NULL, &find_main_arena, NULL);
48         pthread_join(thread, NULL);
49       }
50       else set_main_arena(ptr);
51 }
52
53 //Futex syscall wrapper
54 static inline int futex(int *uaddr, int futex_op, int val,
     const struct timespec *timeout, int *uaddr2, int val3)
55 {
56     return syscall(SYS_futex, uaddr, futex_op, val, timeout,
     uaddr, val3);
57 }
58
59 //Lock an arena
60 static void lock_arena(arena* ar)
61 {
62     int* lock = &ar->lock;
63     if (__sync_val_compare_and_swap(lock,0,1)){
64       do {
65         int old_val = __sync_val_compare_and_swap(lock,1,2);
66         if (old_val != 0) futex(lock, FUTEX_WAIT_PRIVATE, 2,
     NULL, NULL, 0);
67       } while (__sync_val_compare_and_swap(lock,0,2) != 0);
68     }
69
70 }
71
72 //Unlock an arena
73 static void unlock_arena(arena* ar)
```

```
74 {
75   int* lock = &ar->lock;
76   int old_val = __sync_lock_test_and_set(lock,0);
77   if (old_val > 1) futex(lock, FUTEX_WAKE_PRIVATE, 1, NULL,
       NULL,0);
78 }
79
80 bool arena_exists(arena* ar)
81 {
82     int i = 0;
83     while (i < max_arenas && arenas[i] != NULL){
84       if (arenas[i] == ar) return true;
85       i++;
86     }
87     return false;
88 }
89
90 //Add new non-main arena
91 void add_arena(arena* ar)
92 {
93     int i = 0;
94     while (arenas[i] != NULL)
95     {
96       if (arenas[i] == ar) return;
97       i++;
98     }
99     while (__sync_val_compare_and_swap(arenas + i, NULL, ar)
       != NULL)
100    {
101      if (arenas[i] == ar) return;
102      i++;
103    }
104 }
105
106 //Get the amount of used and top memory
107 mem_state get_mem_state()
108 {
109   mem_state state;
110   lock_arena(main_arena);
111   state.top = TOP(main_arena);
112   state.used = main_arena->system_mem;
113   unlock_arena(main_arena);
114   int i = 0;
115   while (i < max_arenas && arenas[i] != NULL)
116   {
117     arena* ar = arenas[i];
```

```
118       lock_arena(ar);
119       state.used += ar->system_mem;
120       state.top += TOP(ar);
121       unlock_arena(ar);
122       i++;
123     }
124     return state;
125   }
126
127   bool need_trim()
128   {
129     size_t top;
130       lock_arena(main_arena);
131       top = TOP(main_arena);
132       unlock_arena(main_arena);
133     int i = 0;
134     while (i < max_arenas && arenas[i] != NULL)
135     {
136       arena* ar = arenas[i];
137       lock_arena(ar);
138       top += TOP(ar);
139       unlock_arena(ar);
140       if (top > TRIM_THRESHOLD) return true;
141       i++;
142     }
143     return false;
144   }
145
146   //Trimming function for non-main arenas
147   static inline void trim_arena(arena* ar)
148   {
149       lock_arena(ar);
150       mchunk_ptr top = ar->top;
151       ptmalloc2_ptr top_chunk = MCHUNK_PTR_TO_PTMALLOC2_PTR(top
          );
152       size_t top_size = SIZE(top_chunk);
153
154       void* addr = (void*)(((long)(top + TOP_PAD) | 4095) + 1);
155       unsigned long len = (unsigned long)(top + top_size - addr
          );
156
157       if (top_size > 2 * TOP_PAD
158           && top_size - len > 32){
159           size_t size_new_top = (top_size - len);
160           madvise(addr, len, MADV_DONTNEED);
161           set_chunk_size_head(top_chunk, size_new_top | 1);
```

```
162
163         heap_info_proxy* top_heap_info = HEAP_INFO((top));
164         top_heap_info->size -= len;
165         ar->system_mem -= len;
166     }
167     unlock_arena(ar);
168 }
169
170 //Trim all arenas
171 void trim_arenas()
172 {
173   malloc_trim(TOP_PAD);
174
175   //Trim non-main arenas
176   int i =0;
177   while ( i < max_arenas && arenas[i] != NULL ){
178     trim_arena(arenas[i]);
179     i++;
180   }
181 }
182
183 int num_arenas()
184 {
185   int i = 0;
186   while (i < max_arenas && arenas[i] != NULL) i++;
187   return i + 1;
188 }
189
190
191 void expand_arena(arena* ar)
192 {
193   lock_arena(ar);
194   heap_info_proxy* info = HEAP_INFO(ar->top);
195   //Recalculate after locking
196   size_t top_mprotect = info->mprotect_size - info->size;
197   if (top_mprotect >= 0.25 * TOP_PAD)
198   {
199     unlock_arena(ar);
200     return;
201   }
202
203   void* addr = (void*)info + info->mprotect_size;
204   size_t len = ((TOP_PAD - top_mprotect) | 4095) + 1;
205   if (info->mprotect_size + len < HEAP_M_SIZE
206       && mprotect(addr, len, PROT_READ | PROT_WRITE) == 0)
207   {
```

```
208       info->mprotect_size += len;
209       ar->system_mem += len;
210       if (ar->system_mem > ar->max_system_mem) ar->
        max_system_mem = ar->system_mem;
211     }
212     else
213     {
214       unlock_arena(ar);
215       return;
216     }
217     unlock_arena(ar);
218   }
```

B.2.4 Ptlbmalloc2.c

```
1  #include <malloc.h>
2  #include <unistd.h>
3  #include <sys/mman.h>
4  #include <stdlib.h>
5  #include <stdio.h>
6
7  #include "global.h"
8  #include "types.h"
9  #include "chunk.h"
10 #include "arena.h"
11 #include "cpu_monitor.h"
12
13
14 //STATIC DATA
15
16 //Synchronization
17 bool init = false;
18 bool init_barrier = false;
19 bool trim_barrier = false;
20
21 //User-controllable sensitivity
22 float tune = 1;
23
24 size_t TOP_PAD = 0;
25 size_t TRIM_THRESHOLD = 100000;
26 size_t HEAP_M_SIZE = 8388608 * sizeof(long);
27 size_t MMAP_THRESHOLD = 128 * 1024;
28 size_t MAX_MMAP_THRESHOLD = 64 * 1024 * 1024;
29
30
31 //Initialization. Executed on first malloc call.
```

```
32 static ptmalloc2_ptr allocate(size_t size, int num){
33   char buf[256];
34   ptmalloc2_ptr ptr;
35   if (!init && __sync_bool_compare_and_swap(&init_barrier,
       false,true)){
36     mallopt(M_TRIM_THRESHOLD, -1);
37     init_cpu_monitor();
38     ptr = num >= 0 ? __libc_calloc(num, size) : __libc_malloc
       (size);
39
40     init_arenas(ptr);
41     init = true;
42   }
43   else {
44     ptr = num >= 0 ? __libc_calloc(num, size) : __libc_malloc
       (size);
45   }
46   if (!IS_MMAPPED(ptr) && init)
47   {
48     size_t size_malloced = SIZE(ptr);
49     if (!MAIN(ptr))
50     {
51       arena* ar = ARENA(ptr);
52       if (!arena_exists(ar)) add_arena(ar);
53       else {
54         heap_info_proxy* info = HEAP_INFO(ptr);
55         arena* ar = info->arena;
56           size_t top_mprotect = info->mprotect_size - info->
       size;
57           if (top_mprotect < 0.25 * TOP_PAD) expand_arena(ar)
       ;
58           }
59     }
60   }
61   return ptr;
62 }
63
64 static inline void update_thresholds()
65 {
66   //Get current memory state
67   mem_state state = get_mem_state();
68   size_t used_size = state.used;
69   size_t top_size = state.top;
70
71   size_t base;
```

```
72    //If allocated memory is smaller than 500kB, use fixed base
        threshold of 100kB
73    if (used_size < 500000)  base = 100000;
74    //If memory is smaller than 1MB, use base threshold of half
        the allocated memory
75    else if (used_size < 1000000) base = 0.5*used_size;
76    //If smaller than 1GB, linearly decrease the percentage of
       memory that the threshold value represents.
77    else if (used_size < 1000000000) base = 0.1 * used_size +
       400000;
78    //If more than 1GB allocated, use fixed threshold of 100MB.
79    else base = 100000000;
80
81    //More CPUs means TLB shootdowns are more expensive, so
       increase threshold based on number of CPUs used.
82    //Allow tuning by user
83    size_t new_trim_threshold = base * (1 + ((float)used_cpus)
       / 100.0) * tune;
84
85    if (new_trim_threshold > 1.25 * TRIM_THRESHOLD
86        || new_trim_threshold < 0.75 * TRIM_THRESHOLD)
87    {
88      TRIM_THRESHOLD = new_trim_threshold;
89      int n_arenas = num_arenas();
90      size_t new_top_pad = new_trim_threshold / 4 / n_arenas;
91      TOP_PAD = new_top_pad;
92      mallopt(M_TOP_PAD, new_top_pad);
93    }
94  }
95
96  //Malloc wrapper
97  void* malloc(size_t size){
98      return allocate(size, -1);
99  }
100
101 //Free wrapper
102 void free(void* ptr){
103     if (ptr != NULL && init)
104     {
105   bool main = MAIN(ptr);
106   bool mmapped = IS_MMAPPED(ptr);
107   size_t size = SIZE(ptr);
108
109   if (mmapped
110       && size > MMAP_THRESHOLD
111       && size <= MAX_MMAP_THRESHOLD)
```

```
112   {
113     MMAP_THRESHOLD = 1.1 * size > MAX_MMAP_THRESHOLD ?
      MAX_MMAP_THRESHOLD : 1.1 * size;
114     mallopt(M_MMAP_THRESHOLD, MMAP_THRESHOLD);
115   }
116
117   arena* ar;
118   size_t old_top_size;
119   if (init && !mmapped) {
120     if (main) ar = main_arena;
121     else {
122             ar = ARENA(ptr);
123       if (!arena_exists(ar)) add_arena(ar);
124     }
125     old_top_size = TOP(ar);
126   }
127
128         __libc_free(ptr);
129
130
131   size_t new_top_size;
132   if (init && !mmapped)
133   {
134     new_top_size = TOP(ar);
135
136     if (new_top_size > old_top_size
137         && new_top_size > 4 * TOP_PAD
138         && !trim_barrier
139         && __sync_bool_compare_and_swap(&trim_barrier, false,
      true))
140     {
141       if (need_trim()){
142         trim_arenas();
143         update_thresholds();
144       }
145       trim_barrier = false;
146           }
147   }
148     }
149     else __libc_free(ptr);
150 }
151
152 void* calloc(size_t num, size_t size){
153   return allocate(size, num);
154 }
155
```

```
156 void* realloc(void* ptr, size_t size){
157     ptmalloc2_ptr mem = __libc_realloc(ptr,size);
158     return mem;
159 }
160
161 //Allow user to control the trade-off between memory
    efficiency and TLB shootdowns
162 //Higher values decrease shootdowns and memory efficiency,
    lower values increase both
163 //Default value is 1
164 //Returns 0 on success, -1 when input is invalid
165 int set_sensitivity(float val){
166     if (val > 0) {
167       tune = val;
168       return 0;
169     }
170     else return -1;
171 }
```

Appendix C

NODedup Source Code

All of the NODedup source files that differ from the original *Dedup* source code upon which it is based.

C.1 Headers

C.1.1 Chunk_list.h

```
1  #ifndef LINKEDLIST_HEADER
2  #define LINKEDLIST_HEADER
3
4  #include <stdio.h>
5  #include <stdlib.h>
6  #include "dedupdef.h"
7
8  typedef struct node{
9    chunk_t * data;
10   struct node * next;
11   int allocated;
12   char used;
13 } Node;
14
15 typedef struct list {
16   Node * head;
17   Node * tail;
18   int length;
19 } List;
```

```
20
21 List * emptylist();
22 void add(chunk_t * elem, List * list);
23 List ** split(int n, List * list);
24 List ** split_mod(int n, List * list);
25 List * merge(List * l1, List * l2);
26 List ** zip_split(int n, List ** lists);
27
28 #endif
```

C.1.2 Iterator.h

```
1 #include<stdlib.h>
2 #include "chunk_list.h"
3
4 typedef struct iterator{
5   List * list;
6   Node * index;
7 }Iterator;
8
9 Iterator * init_iterator(List * list);
10 chunk_t * next(Iterator * iter);
11 Node * next_node(Iterator * iter);
12 void reset(Iterator * iter);
13 int hasNext(Iterator * iter);
14 void destroy_iterator(Iterator * iter);
```

C.1.3 Thread_pool.h

```
1 #ifndef _THPOOL_
2 #define _THPOOL_
3
4 typedef struct thpool_* threadpool;
5
6 threadpool thpool_init(int num_threads);
7 int thpool_add_work(threadpool, void (*function_p)(void*),
      void* arg_p);
8 //Wait for all queued jobs to finish
9 void thpool_wait(threadpool);
10 void thpool_pause(threadpool);
11 void thpool_resume(threadpool);
12 void thpool_destroy(threadpool);
13 int thpool_num_threads_working(threadpool);
14
15 #endif
```

C.1.4 Encoder.h

```
#include "chunk_list.h"

#ifndef _ENCODER_H_
#define _ENCODER_H_ 1

typedef struct{
  size_t size;
  char * data;
} Compressed_data;

void Encode(config_t * conf);

#endif /* !_ENCODER_H_ */
```

C.2 Implementation

C.2.1 Chunk_list.c

```
#include<stdlib.h>
#include<string.h>
#include "chunk_list.h"
#include "iterator.h"

void createNNodes(int n, List * list){
  Node * newNodes = malloc(n * sizeof(Node));

  for (int i = 0; i < n-1; i++){
    newNodes[i].allocated = 0;
    newNodes[i].data = NULL;
    newNodes[i].next = &newNodes[i+1];
    newNodes[i].used = 1;
  }
  newNodes[0].allocated = n;
  newNodes[n-1].next = NULL;
  //No non-empty elemnts
  if(list->tail == NULL) {
    //No empty elements either
    if (list->head == NULL) list->head = newNodes;
    //Only empty elements
    else {
      Node * h = list->head;
      while(h->next != NULL) h = h->next;
      h->next = newNodes;
```

```
26      }
27    }
28    else list->tail->next = newNodes;
29  }
30
31  void createNodes(List * list){
32    int n;
33    if (list->length < 16) n = 16;
34    else if (list->length < 1024) n = list->length;
35    else n = 1024;
36    createNNodes(n,list);
37  }
38
39  List * emptylist(){
40    List * list = malloc(sizeof(List));
41    list->head = NULL;
42    list->tail = NULL;
43    list->length = 0;
44    return list;
45  }
46
47  void add(chunk_t * elem, List * list){
48    if (list->head == NULL) createNodes(list);
49    if (list->head->data == NULL || list->tail == NULL){
50      list->head->data = elem;
51      list->tail = list->head;
52    }
53    else{
54      if(list->tail->next == NULL) createNodes(list);
55      list->tail = list->tail->next;
56      list->tail->data = elem;
57    }
58    list->length++;
59  }
60  void add_node(Node * node, List * list){
61    //Empty list
62    if(list->head == NULL) {
63      node->next = NULL;
64      list->head = node;
65      list->tail = node;
66    }
67    //List with only empty nodes
68    else if (list->head->data == NULL){
69      node->next = list->head;
70      list->head = node;
71      list->tail = node;
```

```
72    }
73    else{
74      node->next = list->tail->next;
75      list->tail->next = node;
76      list->tail = node;
77    }
78    list->length++;
79  }
80
81  //Find the number of memory allocations for a list.
82  int numAllocs(List * list){
83    int len = list->length;
84    if (list == NULL) return 0;
85    else if (len < 17) return 1;
86    else if (len < 33) return 2;
87    else if (len < 65) return 3;
88    else if (len < 129) return 4;
89    else if (len < 257) return 5;
90    else if (len < 513) return 6;
91    else if (len < 1025) return 7;
92    else return 7 + ((len - 1024) / 1024) + ((len % 1024 == 0)
        ? 0:1);
93  }
94
95  //Splits a list in n sublists of sequential elements.
96  List ** split(int n, List * list){
97    int size = (list->length)/n;
98    List ** lists = malloc(n * sizeof(List*));
99    for (int q = 0; q<n;q++) lists[q] = emptylist();
100   Node * buffer;
101   buffer = list->head;
102
103   for(int i = 0; i< list->length; i++){
104     int l;
105     if(i/size < n) l = (i/size);
106     else l = (n-1);
107     Node * nn = buffer->next;
108     add_node(buffer,lists[l]);
109     buffer = nn;
110   }
111   free(list);
112
113   return lists;
114 }
115
```

```
116 //Splits a list in n sublists with each m'th element of the
        sublist being the (n*m)'th element of the original list
117 List ** split_mod(int n, List * list){
118   List ** lists = malloc(n * sizeof(List*));
119   for (int q = 0; q<n;q++) lists[q] = emptylist();
120   int i = 0;
121   Node * buffer = list->head;
122
123   int len = list->length;
124   for(int j = 0; j < len; j++){
125     Node * nn = buffer->next;
126     add_node(buffer,lists[i]);
127     buffer = nn;
128
129     if(i == (n - 1)) i = 0;
130     else i++;
131   }
132   return lists;
133 }
134
135 void merge_empty(List * l1, List * l2){
136   Node * fempty = NULL;
137   Node * lempty = NULL;
138
139   //Initialize fempty;
140   if (l1 == NULL) l1 = emptylist();
141   //Only empty elements in l1
142   else if(l1->head != NULL && l1->tail == NULL) fempty = l1->
        head;
143   //No empty elements in l1
144   if (l1->head == NULL || l1->tail->next == NULL){
145     if (l2 == NULL || l2->head == NULL) return;
146     else if(l2->tail == NULL) fempty = l2->head;
147     else if(l2->tail->next == NULL) return;
148     else fempty = l2->tail->next;
149   }
150   //All other cases
151   else fempty = l1->tail->next;
152
153   //Initialize lempty
154   lempty = fempty;
155   while(lempty->next != NULL) lempty = lempty->next;
156
157   //merge lempty and first empty element of l2 if necessary
```

```
158   if(l2 != NULL && l2->tail != NULL && l2->tail->next != NULL
        && fempty != l2->tail->next) lempty->next = l2->tail->
        next;
159   else if(l2->head != NULL && l2->tail == NULL && fempty !=
        l2->head) lempty->next = l2->head;
160
161   //Do what is necessary to return l1 with the merged empty
        sections;
162   if(l1->head == NULL || l1->tail == NULL) l1->head = fempty;
163   else l1->tail->next = fempty;
164
165   //Remove empty nodes from l2 if necessary
166   if (l2 != NULL && l2->tail != NULL) l2->tail->next = NULL;
167   else if (l2 != NULL && l2->head != NULL && l2->tail == NULL
        ) l2->head = NULL;
168 }
169
170 List * merge(List * l1, List * l2){
171   if (l1 == NULL) return l2;
172   else if (l1->head == NULL){
173     free(l1);
174     return l2;
175   }
176   else if (l2 == NULL) return l1;
177   else if (l2->head == NULL){
178     free(l2);
179     return l1;
180   }
181   merge_empty(l1,l2);
182   l2->tail->next = l1->tail->next;
183   l1->tail->next = l2->head;
184   l1->tail = l2->tail;
185   l1->length += l2->length;
186   free(l2);
187   return l1;
188 }
189
190 //Zips n lists that were split using split_mod.
191 List ** zip_split(int n, List ** lists){
192   List ** output = malloc(n*sizeof(List *));
193   for (int i = 0; i < n; i++){
194     output[i] = emptylist();
195     merge_empty(output[i],lists[i]);
196   }
197   Node * buffers[n];
198   int i;
```

```
199   for(i=0;i<n;i++)buffers[i] = lists[i]->head;
200   int len = lists[0]->length;
201   int out_list = 0;
202   int count = 0;
203
204   for (i = 0; i<len;i++){
205     for (int j = 0; j<n;j++){
206       if(buffers[j]!= NULL){
207         Node * nnn = buffers[j];
208         buffers[j] = nnn->next;
209         add_node(nnn,output[out_list]);
210         count++;
211         if((out_list < n-1) && (count >= len && ((j < n-1 &&
      buffers[j+1] != NULL && buffers[j+1]->data->sequence.
      l1num != nnn->data->sequence.l1num)
212               || (j == n-1 && buffers[0] != NULL && buffers
      [0]->data->sequence.l1num != nnn->data->sequence.l1num)))
      ){
213           out_list++;
214           count = 0;
215         }
216       }
217     }
218   }
219   for(i=0;i<n;i++) free(lists[i]);
220   free(lists);
221   return output;
222 }
```

C.2.2 Iterator.c

```
1  #include "iterator.h"
2
3  Iterator * init_iterator(List * list){
4    Iterator * iter = malloc(sizeof(Iterator));
5    iter->list = list;
6    iter->index = NULL;
7    return iter;
8  }
9
10 Node * next_node(Iterator * iter){
11   Node * n = iter->index;
12   if (n == NULL) {
13     n = iter->list->head;
14     if (n == NULL || n->data==NULL) return NULL;
15     else{
```

```
16        iter->index = n;
17        return n;
18      }
19    }
20    Node * nn = n->next;
21    if(nn == NULL || nn->data==NULL) return NULL;
22    iter->index = nn;
23    return nn;
24  }
25
26  chunk_t * next(Iterator * iter){
27    Node * n = next_node(iter);
28    if (n == NULL || n->data == NULL) return NULL;
29    return n->data;
30  };
31
32  void reset(Iterator * iter){
33      iter->index = NULL;
34  }
35
36  int hasNext(Iterator * iter){
37    return (iter->index != iter->list->tail);
38  }
39
40  void destroy_iterator(Iterator * iter){
41    free(iter);
42  }
```

C.2.3 Thread_pool.c

```
1  /**********************************
2   * @author       Johan Hanssen Seferidis
3   * License:     MIT
4   *
5   **********************************/
6
7  /* ********************************
8   * Author:        Johan Hanssen Seferidis
9   * License:       MIT
10  * Description:  Library providing a threading pool where you
      can add
11  *               work. For usage, check the thpool.h file or
      README.md
12  *
13  *//** @file thpool.h *//*
14  *
```

```
15   *******************************/
16
17  #define _POSIX_C_SOURCE 200809L
18  #include <unistd.h>
19  #include <signal.h>
20  #include <stdio.h>
21  #include <stdlib.h>
22  #include <pthread.h>
23  #include <errno.h>
24  #include <time.h>
25  #if defined(__linux__)
26  #include <sys/prctl.h>
27  #endif
28  #include "thpool.h"
29  #ifdef THPOOL_DEBUG
30  #define THPOOL_DEBUG 1
31  #else
32  #define THPOOL_DEBUG 0
33  #endif
34  #if !defined(DISABLE_PRINT) || defined(THPOOL_DEBUG)
35  #define err(str) fprintf(stderr, str)
36  #else
37  #define err(str)
38  #endif
39
40  static volatile int threads_keepalive;
41  static volatile int threads_on_hold;
42
43  typedef struct bsem {
44    pthread_mutex_t mutex;
45    pthread_cond_t  cond;
46    int v;
47  } bsem;
48
49  typedef struct job{
50    struct job*  prev;
51    void   (*function)(void* arg);
52    void*  arg;
53  } job;
54
55  typedef struct jobqueue{
56    pthread_mutex_t rwmutex;
57    job  *front;
58    job  *rear;
59    bsem *has_jobs;
60    int   len;
```

```
61 } jobqueue;
62
63 typedef struct thread{
64   int        id;
65   pthread_t pthread;
66   struct thpool_* thpool_p;
67 } thread;
68
69 typedef struct thpool_{
70   thread**   threads;
71   volatile int num_threads_alive;
72   volatile int num_threads_working;
73   pthread_mutex_t  thcount_lock;
74   pthread_cond_t  threads_all_idle;
75   jobqueue  jobqueue;
76 } thpool_;
77
78 static int  thread_init(thpool_* thpool_p, struct thread**
       thread_p, int id);
79 static void* thread_do(struct thread* thread_p);
80 static void  thread_hold(int sig_id);
81 static void  thread_destroy(struct thread* thread_p);
82
83 static int   jobqueue_init(jobqueue* jobqueue_p);
84 static void  jobqueue_clear(jobqueue* jobqueue_p);
85 static void  jobqueue_push(jobqueue* jobqueue_p, struct job*
       newjob_p);
86 static struct job* jobqueue_pull(jobqueue* jobqueue_p);
87 static void  jobqueue_destroy(jobqueue* jobqueue_p);
88
89 static void  bsem_init(struct bsem *bsem_p, int value);
90 static void  bsem_reset(struct bsem *bsem_p);
91 static void  bsem_post(struct bsem *bsem_p);
92 static void  bsem_post_all(struct bsem *bsem_p);
93 static void  bsem_wait(struct bsem *bsem_p);
94
95 /* Initialise thread pool */
96 struct thpool_* thpool_init(int num_threads){
97
98   threads_on_hold   = 0;
99   threads_keepalive = 1;
100
101   if (num_threads < 0){
102     num_threads = 0;
103   }
104
```

```
105   /* Make new thread pool */
106   thpool_* thpool_p;
107   thpool_p = (struct thpool_*)malloc(sizeof(struct thpool_));
108   if (thpool_p == NULL){
109     err("thpool_init(): Could not allocate memory for thread
      pool\n");
110     return NULL;
111   }
112   thpool_p->num_threads_alive   = 0;
113   thpool_p->num_threads_working = 0;
114
115   /* Initialise the job queue */
116   if (jobqueue_init(&thpool_p->jobqueue) == -1){
117     err("thpool_init(): Could not allocate memory for job
      queue\n");
118     free(thpool_p);
119     return NULL;
120   }
121
122   /* Make threads in pool */
123   thpool_p->threads = (struct thread**)malloc(num_threads *
      sizeof(struct thread *));
124   if (thpool_p->threads == NULL){
125     err("thpool_init(): Could not allocate memory for threads
      \n");
126     jobqueue_destroy(&thpool_p->jobqueue);
127     free(thpool_p);
128     return NULL;
129   }
130
131   pthread_mutex_init(&(thpool_p->thcount_lock), NULL);
132   pthread_cond_init(&thpool_p->threads_all_idle, NULL);
133
134   /* Thread init */
135   int n;
136   for (n=0; n<num_threads; n++){
137     thread_init(thpool_p, &thpool_p->threads[n], n);
138 #if THPOOL_DEBUG
139       printf("THPOOL_DEBUG: Created thread %d in pool \n", n)
      ;
140 #endif
141   }
142
143   /* Wait for threads to initialize */
144   while (thpool_p->num_threads_alive != num_threads) {}
145
```

```
146   return thpool_p;
147 }
148
149
150 /* Add work to the thread pool */
151 int thpool_add_work(thpool_* thpool_p, void (*function_p)(
      void*), void* arg_p){
152   job* newjob;
153
154   newjob=(struct job*)malloc(sizeof(struct job));
155   if (newjob==NULL){
156     err("thpool_add_work(): Could not allocate memory for new
       job\n");
157     return -1;
158   }
159
160   /* add function and argument */
161   newjob->function=function_p;
162   newjob->arg=arg_p;
163
164   /* add job to queue */
165   jobqueue_push(&thpool_p->jobqueue, newjob);
166
167   return 0;
168 }
169
170
171 /* Wait until all jobs have finished */
172 void thpool_wait(thpool_* thpool_p){
173   pthread_mutex_lock(&thpool_p->thcount_lock);
174   while (thpool_p->jobqueue.len || thpool_p->
      num_threads_working) {
175     pthread_cond_wait(&thpool_p->threads_all_idle, &thpool_p
      ->thcount_lock);
176   }
177   pthread_mutex_unlock(&thpool_p->thcount_lock);
178 }
179
180
181 /* Destroy the threadpool */
182 void thpool_destroy(thpool_* thpool_p){
183   /* No need to destory if it's NULL */
184   if (thpool_p == NULL) return ;
185
186   volatile int threads_total = thpool_p->num_threads_alive;
187
```

```
188   /* End each thread 's infinite loop */
189   threads_keepalive = 0;
190
191   /* Give one second to kill idle threads */
192   double TIMEOUT = 1.0;
193   time_t start, end;
194   double tpassed = 0.0;
195   time (&start);
196   while (tpassed < TIMEOUT && thpool_p->num_threads_alive){
197     bsem_post_all(thpool_p->jobqueue.has_jobs);
198     time (&end);
199     tpassed = difftime(end,start);
200   }
201
202   /* Poll remaining threads */
203   while (thpool_p->num_threads_alive){
204     bsem_post_all(thpool_p->jobqueue.has_jobs);
205     sleep(1);
206   }
207
208   /* Job queue cleanup */
209   jobqueue_destroy(&thpool_p->jobqueue);
210   /* Deallocs */
211   int n;
212   for (n=0; n < threads_total; n++){
213     thread_destroy(thpool_p->threads[n]);
214   }
215   free(thpool_p->threads);
216   free(thpool_p);
217 }
218
219
220 /* Pause all threads in threadpool */
221 void thpool_pause(thpool_* thpool_p) {
222   int n;
223   for (n=0; n < thpool_p->num_threads_alive; n++){
224     pthread_kill(thpool_p->threads[n]->pthread, SIGUSR1);
225   }
226 }
227
228
229 /* Resume all threads in threadpool */
230 void thpool_resume(thpool_* thpool_p) {
231     // resuming a single threadpool hasn't been
232     // implemented yet, meanwhile this supresses
233     // the warnings
```

```
234     (void)thpool_p;
235 
236   threads_on_hold = 0;
237 }
238 
239 
240 int thpool_num_threads_working(thpool_* thpool_p){
241   return thpool_p->num_threads_working;
242 }
243 
244 /* Initialize a thread in the thread pool
245  *
246  * @param thread        address to the pointer of the thread
       to be created
247  * @param id            id to be given to the thread
248  * @return 0 on success, -1 otherwise.
249  */
250 static int thread_init (thpool_* thpool_p, struct thread**
       thread_p, int id){
251 
252   *thread_p = (struct thread*)malloc(sizeof(struct thread));
253   if (thread_p == NULL){
254     err("thread_init(): Could not allocate memory for thread\
      n");
255     return -1;
256   }
257 
258   (*thread_p)->thpool_p = thpool_p;
259   (*thread_p)->id       = id;
260 
261   pthread_create(&(*thread_p)->pthread, NULL, (void *)
       thread_do, (*thread_p));
262   pthread_detach((*thread_p)->pthread);
263   return 0;
264 }
265 
266 /* Sets the calling thread on hold */
267 static void thread_hold(int sig_id) {
268     (void)sig_id;
269   threads_on_hold = 1;
270   while (threads_on_hold){
271     sleep(1);
272   }
273 }
274 
275 
```

```
276 /* What each thread is doing
277 *
278 * In principle this is an endless loop. The only time this
      loop gets interuppted is once
279 * thpool_destroy() is invoked or the program exits.
280 *
281 * @param  thread        thread that will run this function
282 * @return nothing
283 */
284 static void* thread_do(struct thread* thread_p){
285
286   /* Set thread name for profiling and debuging */
287   char thread_name[128] = {0};
288   sprintf(thread_name, "thread-pool-%d", thread_p->id);
289
290 #if defined(__linux__)
291   /* Use prctl instead to prevent using _GNU_SOURCE flag and
      implicit declaration */
292   prctl(PR_SET_NAME, thread_name);
293 #elif defined(__APPLE__) && defined(__MACH__)
294   pthread_setname_np(thread_name);
295 #else
296   err("thread_do(): pthread_setname_np is not supported on
      this system");
297 #endif
298
299   /* Assure all threads have been created before starting
      serving */
300   thpool_* thpool_p = thread_p->thpool_p;
301
302   /* Register signal handler */
303   struct sigaction act;
304   sigemptyset(&act.sa_mask);
305   act.sa_flags = 0;
306   act.sa_handler = thread_hold;
307   if (sigaction(SIGUSR1, &act, NULL) == -1) {
308     err("thread_do(): cannot handle SIGUSR1");
309   }
310
311   /* Mark thread as alive (initialized) */
312   pthread_mutex_lock(&thpool_p->thcount_lock);
313   thpool_p->num_threads_alive += 1;
314   pthread_mutex_unlock(&thpool_p->thcount_lock);
315
316   while(threads_keepalive){
317
```

```
318       bsem_wait(thpool_p->jobqueue.has_jobs);
319
320       if (threads_keepalive){
321
322         pthread_mutex_lock(&thpool_p->thcount_lock);
323         thpool_p->num_threads_working++;
324         pthread_mutex_unlock(&thpool_p->thcount_lock);
325
326         /* Read job from queue and execute it */
327         void (*func_buff)(void*);
328         void*  arg_buff;
329         job* job_p = jobqueue_pull(&thpool_p->jobqueue);
330         if (job_p) {
331           func_buff = job_p->function;
332           arg_buff  = job_p->arg;
333           func_buff(arg_buff);
334           free(job_p);
335         }
336
337         pthread_mutex_lock(&thpool_p->thcount_lock);
338         thpool_p->num_threads_working--;
339         if (!thpool_p->num_threads_working) {
340           pthread_cond_signal(&thpool_p->threads_all_idle);
341         }
342         pthread_mutex_unlock(&thpool_p->thcount_lock);
343
344       }
345     }
346     pthread_mutex_lock(&thpool_p->thcount_lock);
347     thpool_p->num_threads_alive --;
348     pthread_mutex_unlock(&thpool_p->thcount_lock);
349
350     return NULL;
351   }
352
353   /* Frees a thread  */
354   static void thread_destroy (thread* thread_p){
355     free(thread_p);
356   }
357
358   /* Initialize queue */
359   static int jobqueue_init(jobqueue* jobqueue_p){
360     jobqueue_p->len = 0;
361     jobqueue_p->front = NULL;
362     jobqueue_p->rear  = NULL;
363
```

```
364   jobqueue_p->has_jobs = (struct bsem*)malloc(sizeof(struct
        bsem));
365   if (jobqueue_p->has_jobs == NULL){
366     return -1;
367   }
368
369   pthread_mutex_init(&(jobqueue_p->rwmutex), NULL);
370   bsem_init(jobqueue_p->has_jobs, 0);
371
372   return 0;
373 }
374
375 /* Clear the queue */
376 static void jobqueue_clear(jobqueue* jobqueue_p){
377
378   while(jobqueue_p->len)
379     free(jobqueue_pull(jobqueue_p));
380
381   jobqueue_p->front = NULL;
382   jobqueue_p->rear  = NULL;
383   bsem_reset(jobqueue_p->has_jobs);
384   jobqueue_p->len = 0;
385
386 }
387
388
389 // Add (allocated) job to queue
390 static void jobqueue_push(jobqueue* jobqueue_p, struct job*
      newjob){
391
392   pthread_mutex_lock(&jobqueue_p->rwmutex);
393   newjob->prev = NULL;
394
395   switch(jobqueue_p->len){
396
397     case 0:  /* if no jobs in queue */
398           jobqueue_p->front = newjob;
399           jobqueue_p->rear  = newjob;
400           break;
401
402     default: /* if jobs in queue */
403           jobqueue_p->rear->prev = newjob;
404           jobqueue_p->rear = newjob;
405
406   }
407   jobqueue_p->len++;
```

```
408
409   bsem_post(jobqueue_p->has_jobs);
410   pthread_mutex_unlock(&jobqueue_p->rwmutex);
411 }
412
413
414 // Get first job from queue(removes it from queue)
415 static struct job* jobqueue_pull(jobqueue* jobqueue_p){
416
417   pthread_mutex_lock(&jobqueue_p->rwmutex);
418   job* job_p = jobqueue_p->front;
419
420   switch(jobqueue_p->len){
421
422     case 0:  /* if no jobs in queue */
423               break;
424
425     case 1:  /* if one job in queue */
426           jobqueue_p->front = NULL;
427           jobqueue_p->rear  = NULL;
428           jobqueue_p->len = 0;
429           break;
430
431     default: /* if >1 jobs in queue */
432           jobqueue_p->front = job_p->prev;
433           jobqueue_p->len--;
434           /* more than one job in queue -> post it */
435           bsem_post(jobqueue_p->has_jobs);
436
437   }
438
439   pthread_mutex_unlock(&jobqueue_p->rwmutex);
440   return job_p;
441 }
442
443 /* Free all queue resources back to the system */
444 static void jobqueue_destroy(jobqueue* jobqueue_p){
445   jobqueue_clear(jobqueue_p);
446   free(jobqueue_p->has_jobs);
447 }
448
449 /* Init semaphore to 1 or 0 */
450 static void bsem_init(bsem *bsem_p, int value) {
451   if (value < 0 || value > 1) {
452     err("bsem_init(): Binary semaphore can take only values 1
       or 0");
```

```
453       exit(1);
454     }
455     pthread_mutex_init(&(bsem_p->mutex), NULL);
456     pthread_cond_init(&(bsem_p->cond), NULL);
457     bsem_p->v = value;
458   }
459
460   /* Reset semaphore to 0 */
461   static void bsem_reset(bsem *bsem_p) {
462     bsem_init(bsem_p, 0);
463   }
464
465   /* Post to at least one thread */
466   static void bsem_post(bsem *bsem_p) {
467     pthread_mutex_lock(&bsem_p->mutex);
468     bsem_p->v = 1;
469     pthread_cond_signal(&bsem_p->cond);
470     pthread_mutex_unlock(&bsem_p->mutex);
471   }
472
473   /* Post to all threads */
474   static void bsem_post_all(bsem *bsem_p) {
475     pthread_mutex_lock(&bsem_p->mutex);
476     bsem_p->v = 1;
477     pthread_cond_broadcast(&bsem_p->cond);
478     pthread_mutex_unlock(&bsem_p->mutex);
479   }
480
481
482   /* Wait on semaphore until semaphore has value 0 */
483   static void bsem_wait(bsem* bsem_p) {
484     pthread_mutex_lock(&bsem_p->mutex);
485     while (bsem_p->v != 1) {
486       pthread_cond_wait(&bsem_p->cond, &bsem_p->mutex);
487     }
488     bsem_p->v = 0;
489     pthread_mutex_unlock(&bsem_p->mutex);
490   }
```

C.2.4 Encoder.c

```
1  /*
2   * Decoder for dedup files
3   *
4   * Copyright 2010 Princeton University.
5   * All rights reserved.
```

```
 *
 * Originally written by Minlan Yu.
 * Largely rewritten by Christian Bienia.
*/

/*
 * The pipeline model for Encode is Fragment->FragmentRefine
   ->Deduplicate->Compress->Reorder
 * Each stage has basically three steps:
 * 1. fetch a group of items from the queue
 * 2. process the items
 * 3. put them in the queue for the next stage
*/

#include <assert.h>
#include <strings.h>
#include <math.h>
#include <limits.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <errno.h>
#include <unistd.h>
#include <string.h>
#include "util.h"
#include "dedupdef.h"
#include "encoder.h"
#include "debug.h"
#include "hashtable.h"
#include "config.h"
#include "rabin.h"
#include "mbuffer.h"
#include "chunk_list.h"
#include "iterator.h"
#include "thpool.h"
#ifdef ENABLE_PTHREADS
#include "binheap.h"
#include "tree.h"
#endif //ENABLE_PTHREADS
#ifdef ENABLE_GZIP_COMPRESSION
#include <zlib.h>
#endif //ENABLE_GZIP_COMPRESSION
#ifdef ENABLE_BZIP2_COMPRESSION
#include <bzlib.h>
#endif //ENABLE_BZIP2_COMPRESSION
#ifdef ENABLE_PTHREADS
#include <pthread.h>
```

```
51 #endif //ENABLE_PTHREADS
52 #ifdef ENABLE_PARSEC_HOOKS
53 #include <hooks.h>
54 #endif //ENABLE_PARSEC_HOOKS
55
56
57 #define INITIAL_SEARCH_TREE_SIZE 4096
58
59
60 //The configuration block defined in main
61 config_t * conf;
62 //Hash table data structure & utility functions
63 struct hashtable *cache;
64 static unsigned int hash_from_key_fn( void *k ) {
65   //NOTE: sha1 sum is integer-aligned
66   return ((unsigned int *)k)[0];
67 }
68 static int keys_equal_fn ( void *key1, void *key2 ) {
69   return (memcmp(key1, key2, SHA1_LEN) == 0);
70 }
71
72
73 #ifdef ENABLE_STATISTICS
74 //Keep track of block granularity
75 #define CHUNK_GRANULARITY_POW (7)
76 //Number of blocks to distinguish
77 #define CHUNK_MAX_NUM (8*32)
78 //Map a chunk size to a statistics array slot
79 #define CHUNK_SIZE_TO_SLOT(s) ( ((s)>>(CHUNK_GRANULARITY_POW)
       ) >= (CHUNK_MAX_NUM) ? (CHUNK_MAX_NUM)-1 : ((s)>>(
       CHUNK_GRANULARITY_POW)) )
80 //Get the average size of a chunk from a statistics array
       slot
81 #define SLOT_TO_CHUNK_SIZE(s) ( (s)*(1<<(
       CHUNK_GRANULARITY_POW)) + (1<<((CHUNK_GRANULARITY_POW)-1)
       ) )
82
83
84 //Deduplication statistics
85 typedef struct {
86   /* Cumulative sizes */
87   size_t total_input; //Total size of input in bytes
88   size_t total_dedup; //Total size of input without duplicate
       blocks (after global compression) in bytes
89   size_t total_compressed; //Total size of input stream after
       local compression in bytes
```

```
90    size_t total_output; //Total size of output in bytes (with
        overhead) in bytes
91
92    /* Size distribution & other properties */
93    unsigned int nChunks[CHUNK_MAX_NUM]; //Coarse-granular size
        distribution of data chunks
94    unsigned int nDuplicates; //Total number of duplicate
        blocks
95  } stats_t;
96
97  //Arguments to pass to each thread
98  struct thread_args {
99    //thread id, unique within a thread pool (i.e. unique for a
        pipeline stage)
100   int tid;
101   //number of queues available, first and last pipeline stage
        only
102   int nqueues;
103   //file descriptor, first pipeline stage only
104   int fd;
105   //List of chunks
106   List * list;
107
108   //char ** compressed_data;
109   Compressed_data * compressed_data;
110
111   List ** list_addr;
112   //input file buffer, first pipeline stage & preloading only
113   struct {
114     void *buffer;
115     size_t size;
116   } input_file;
117
118   stats_t * stats;
119 };
120
121 //Initialize a statistics record
122 static void init_stats(stats_t *s) {
123   int i;
124
125   assert(s!=NULL);
126   s->total_input = 0;
127   s->total_dedup = 0;
128   s->total_compressed = 0;
129   s->total_output = 0;
130
```

```
131   for(i=0; i<CHUNK_MAX_NUM; i++) {
132     s->nChunks[i] = 0;
133   }
134   s->nDuplicates = 0;
135 }
136
137 #ifdef ENABLE_PTHREADS
138
139 //Merge two statistics records: s1=s1+s2
140 static void merge_stats(stats_t *s1, stats_t *s2) {
141   int i;
142
143   assert(s1!=NULL);
144   assert(s2!=NULL);
145   s1->total_input += s2->total_input;
146   s1->total_dedup += s2->total_dedup;
147   s1->total_compressed += s2->total_compressed;
148   s1->total_output += s2->total_output;
149
150   for(i=0; i<CHUNK_MAX_NUM; i++) {
151     s1->nChunks[i] += s2->nChunks[i];
152   }
153   s1->nDuplicates += s2->nDuplicates;
154 }
155 #endif //ENABLE_PTHREADS
156
157 //Print statistics
158 static void print_stats(stats_t *s) {
159   const unsigned int unit_str_size = 7; //elements in
        unit_str array
160   const char *unit_str[] = {"Bytes", "KB", "MB", "GB", "TB",
        "PB", "EB"};
161   unsigned int unit_idx = 0;
162   size_t unit_div = 1;
163
164   assert(s!=NULL);
165
166   //determine most suitable unit to use
167   for(unit_idx=0; unit_idx<unit_str_size; unit_idx++) {
168     unsigned int unit_div_next = unit_div * 1024;
169
170     if(s->total_input / unit_div_next <= 0) break;
171     if(s->total_dedup / unit_div_next <= 0) break;
172     if(s->total_compressed / unit_div_next <= 0) break;
173     if(s->total_output / unit_div_next <= 0) break;
174
```

```
175       unit_div = unit_div_next;
176     }
177
178     printf("Total input size:                %14.2f %s\n", (float
          )(s->total_input)/(float)(unit_div), unit_str[unit_idx]);
179     printf("Total output size:               %14.2f %s\n", (float
          )(s->total_output)/(float)(unit_div), unit_str[unit_idx])
          ;
180     printf("Effective compression factor:  %14.2fx\n", (float)(
          s->total_input)/(float)(s->total_output));
181     printf("\n");
182
183     //Total number of chunks
184     unsigned int i;
185     unsigned int nTotalChunks=0;
186     for(i=0; i<CHUNK_MAX_NUM; i++) nTotalChunks+= s->nChunks[i
          ];
187
188     //Average size of chunks
189     float mean_size = 0.0;
190     for(i=0; i<CHUNK_MAX_NUM; i++) mean_size += (float)(
          SLOT_TO_CHUNK_SIZE(i)) * (float)(s->nChunks[i]);
191     mean_size = mean_size / (float)nTotalChunks;
192
193     //Variance of chunk size
194     float var_size = 0.0;
195     for(i=0; i<CHUNK_MAX_NUM; i++) var_size += (mean_size - (
          float)(SLOT_TO_CHUNK_SIZE(i))) *
196                                                (mean_size - (
          float)(SLOT_TO_CHUNK_SIZE(i))) *
197                                                (float)(s->
          nChunks[i]);
198
199     printf("Total number of chunks: %d, Duplicate chunks: %d\n"
          ,nTotalChunks,s->nDuplicates);
200
201     printf("Mean data chunk size:            %14.2f %s (stddev:
          %.2f %s)\n", mean_size / 1024.0, "KB", sqrtf(var_size) /
          1024.0, "KB");
202     printf("Amount of duplicate chunks:    %14.2f%%\n", 100.0*(
          float)(s->nDuplicates)/(float)(nTotalChunks));
203     printf("Data size after deduplication: %14.2f %s (
          compression factor: %.2fx)\n", (float)(s->total_dedup)/(
          float)(unit_div), unit_str[unit_idx], (float)(s->
          total_input)/(float)(s->total_dedup));
```

```
204   printf("Data size after compression:   %14.2f %s (
        compression factor: %.2fx)\n", (float)(s->
        total_compressed)/(float)(unit_div), unit_str[unit_idx],
        (float)(s->total_dedup)/(float)(s->total_compressed));
205   printf("Output overhead:               %14.2f%%\n", 100.0*(
        float)(s->total_output-s->total_compressed)/(float)(s->
        total_output));
206 }
207
208 //variable with global statistics
209 stats_t stats;
210 #endif //ENABLE_STATISTICS
211
212 /*
213  * Helper function that creates and initializes the output
        file
214  * Takes the file name to use as input and returns the file
        handle
215  * The output file can be used to write chunks without any
        further steps
216  */
217 static int create_output_file(char *outfile) {
218   int fd;
219
220   //Create output file
221   fd = open(outfile, O_CREAT|O_TRUNC|O_WRONLY|O_TRUNC,
        S_IRGRP | S_IWUSR | S_IRUSR | S_IROTH);
222   if (fd < 0) {
223     EXIT_TRACE("Cannot open output file.");
224   }
225
226   //Write header
227   if (write_header(fd, conf->compress_type)) {
228     EXIT_TRACE("Cannot write output file header.\n");
229   }
230   return fd;
231 }
232
233 int rf_win;
234 int rf_win_dataprocess;
235
236 /*
237  * Computational kernel of compression stage
238  * Actions performed: Compress a data chunk
239  */
240 void sub_Compress(chunk_t *chunk) {
```

```
241     int r;
242
243     assert(chunk!=NULL);
244     switch (conf->compress_type) {
245       case COMPRESS_NONE:
246         //copy the block
247         chunk->compressed_data.n = chunk->uncompressed_data.n
    ;
248         memcpy(chunk->compressed_data.ptr, chunk->
    uncompressed_data.ptr, chunk->uncompressed_data.n);
249         break;
250 #ifdef ENABLE_GZIP_COMPRESSION
251       case COMPRESS_GZIP:
252         r = compress(chunk->compressed_data.ptr, &chunk->
    compressed_data.n, chunk->uncompressed_data.ptr, chunk->
    uncompressed_data.n);
253         if (r != Z_OK) {
254           EXIT_TRACE("Compression failed. Error code: %d\n",r
    );
255         }
256         break;
257 #endif //ENABLE_GZIP_COMPRESSION
258 #ifdef ENABLE_BZIP2_COMPRESSION
259       case COMPRESS_BZIP2:
260         //Bzip compression buffer must be at least 1% larger
    than source buffer plus 600 bytes
261         n = chunk->uncompressed_data.n + (chunk->
    uncompressed_data.n >> 6) + 600;
262         r = mbuffer_create(&chunk->compressed_data, n);
263         if(r != 0) {
264           EXIT_TRACE("Creation of compression buffer failed.\
    n");
265         }
266         //compress the block
267         unsigned int int_n = n;
268         r = BZ2_bzBuffToBuffCompress(chunk->compressed_data.
    ptr, &int_n, chunk->uncompressed_data.ptr, chunk->
    uncompressed_data.n, 9, 0, 30);
269         n = int_n;
270         if (r != BZ_OK) {
271           EXIT_TRACE("Compression failed\n");
272         }
273         //Shrink buffer to actual size
274         if(n < chunk->compressed_data.n) {
275           r = mbuffer_realloc(&chunk->compressed_data, n);
276           assert(r == 0);
```

```
277         }
278         break;
279     #endif //ENABLE_BZIP2_COMPRESSION
280       default:
281         EXIT_TRACE("Compression type not implemented.\n");
282         break;
283     }
284     mbuffer_free(&chunk->uncompressed_data);
285
286 #ifdef ENABLE_PTHREADS
287     chunk->header.state = CHUNK_STATE_COMPRESSED;
288 #endif //ENABLE_PTHREADS
289
290 }
291
292 /*
293  * Pipeline stage function of compression stage
294  *
295  * Actions performed:
296  *  - Dequeue items from compression queue
297  *  - Execute compression kernel for each item
298  *  - Enqueue each item into send queue
299  */
300 //#ifdef ENABLE_PTHREADS
301 void Compress(void * targs) {
302
303   struct thread_args *args = (struct thread_args *)targs;
304   List * list = args->list;
305
306   #ifdef ENABLE_STATISTICS
307   stats_t * thread_stats = args->stats;
308   init_stats(thread_stats);
309   #endif //ENABLE_STATISTICS
310
311   //Allocate memory for compressed data buffers
312
313   int total_chunks = 0;
314   int duplicate_chunks = 0;
315   size_t total_size = 0;
316   void * mbuffers;
317   chunk_t * chunk_refs[1000];
318
319   int write_buffers_index = 0;
320
321   Iterator * iter = init_iterator(list);
322   while(hasNext(iter)){
```

```
323     chunk_t * c = next(iter);
324     chunk_refs[total_chunks] = c;
325     total_chunks++;
326
327     // If chunk is unique, update counter to reserve memory
        for compressed buffer.
328     if(c->header.isDuplicate) duplicate_chunks++;
329     else{
330       thread_stats->total_dedup += c->uncompressed_data.n;
331       size_t * size = &c->compressed_data.n;
332       if(conf->compress_type == COMPRESS_NONE) *size =  c->
        uncompressed_data.n;
333       else *size = c->uncompressed_data.n + (c->
        uncompressed_data.n >> 9) + 12;
334       total_size += *size;
335     }
336
337     //If we found 1000 chunks or found the last chunk,
        process the batch.
338     if(total_chunks == 1000 || !hasNext(iter)){
339       int index = 0;
340       mbuffers = malloc(total_size);
341       total_size = 0;
342       for(int i = 0; i<total_chunks; i++){
343         chunk_t * chunk = chunk_refs[i];
344         if(!chunk->header.isDuplicate){
345           chunk->compressed_data.ptr = mbuffers + index;
346           index += chunk->compressed_data.n;
347           sub_Compress(chunk);
348           thread_stats->total_compressed += chunk->
        compressed_data.n;
349           total_size += chunk->compressed_data.n;
350         }
351       }
352
353       int write_buffer_size = duplicate_chunks * SHA1_LEN +
        total_chunks * 9 + total_size;
354       char * write_buffer = malloc(write_buffer_size);
355
356       index = 0;
357       for(int i = 0; i < total_chunks; i++){
358         chunk_t * chunk = chunk_refs[i];
359         if (chunk->header.isDuplicate){
360           thread_stats->nDuplicates++;
361           write_buffer[index] = 0;
362           *((u_long*) (&write_buffer[index+1])) = SHA1_LEN;
```

```
363             index += 9;
364             memcpy(write_buffer + index, &chunk->sha1, SHA1_LEN
    );
365             index += SHA1_LEN;
366           }
367           else{
368             write_buffer[index] = 1;
369             *((u_long*) (&write_buffer[index+1])) = chunk->
    compressed_data.n;
370             index += 9;
371             memcpy(write_buffer + index, chunk->compressed_data
    .ptr, chunk->compressed_data.n);
372             index += chunk->compressed_data.n;
373           }
374           free(chunk);
375         }
376         free(mbuffers);
377         args->compressed_data[write_buffers_index].data =
    write_buffer;
378         args->compressed_data[write_buffers_index].size =
    write_buffer_size;
379         write_buffers_index++;
380         total_chunks = 0;
381         duplicate_chunks = 0;
382       }
383     }
384     destroy_iterator(iter);
385     free(list);
386 }
387
388  /* Computational kernel of deduplication stage
389  *
390  * Actions performed:
391  *  - Calculate SHA1 signature for each incoming data chunk
392  *  - Perform database lookup to determine chunk redundancy
    status
393  *  - On miss add chunk to database
394  *  - Returns chunk redundancy status */
395 int sub_Deduplicate(chunk_t *chunk) {
396   int isDuplicate;
397   int isFirst = 1;
398   chunk_t *entry;
399
400   assert(chunk!=NULL);
401   assert(chunk->uncompressed_data.ptr!=NULL);
402
```

```
403   SHA1_Digest(chunk->uncompressed_data.ptr, chunk->
        uncompressed_data.n, (unsigned char *)(chunk->sha1));
404
405   //Query database to determine whether we've seen the data
        chunk before
406 #ifdef ENABLE_PTHREADS
407   pthread_mutex_t *ht_lock = hashtable_getlock(cache, (void
        *)(chunk->sha1));
408   pthread_mutex_lock(ht_lock);
409 #endif
410   entry = (chunk_t *)hashtable_search(cache, (void *)(chunk->
        sha1));
411   isDuplicate = (entry != NULL);
412   if (isDuplicate){
413     if (entry->sequence.l1num > chunk->sequence.l1num
414       || (entry->sequence.l1num == chunk->sequence.l1num
415         && entry->sequence.l2num > chunk->sequence.l2num)){
416       isFirst = 1;
417       entry->header.isDuplicate = 1;
418       chunk->header.isDuplicate = 0;
419       entry->compressed_data_ref = chunk;
420       mbuffer_free(&entry->uncompressed_data);
421       if (hashtable_insert(cache, (void *)(chunk->sha1), (
        void *)chunk) == 0) {
422         EXIT_TRACE("hashtable_insert failed");
423       }
424     }
425     else{
426       isFirst = 0;
427       chunk->header.isDuplicate = 1;
428       entry->header.isDuplicate = 0;
429       chunk->compressed_data_ref = entry;
430       mbuffer_free(&chunk->uncompressed_data);
431     }
432   }
433   else{
434     chunk->header.isDuplicate = 0;
435     // Cache miss: Create entry in hash table and forward
        data to compression stage
436     #ifdef ENABLE_PTHREADS
437       pthread_mutex_init(&chunk->header.lock, NULL);
438       pthread_cond_init(&chunk->header.update, NULL);
439     #endif
440       //NOTE: chunk->compressed_data.buffer will be computed
        in compression stage
```

```
441     if (hashtable_insert(cache, (void *)(chunk->sha1), (void
        *)chunk) == 0) {
442         EXIT_TRACE("hashtable_insert failed");
443     }
444   }
445 #ifdef ENABLE_PTHREADS
446   pthread_mutex_unlock(ht_lock);
447 #endif
448
449   return (isDuplicate && !isFirst);
450 }
451
452 /* Pipeline stage function of deduplication stage
453  *
454  * Actions performed:
455  *  - Take input data from fragmentation stages
456  *  - Execute deduplication kernel for each data chunk
457  *  - Route resulting package either to compression stage or
       to reorder stage, depending on deduplication status */
458 #ifdef ENABLE_PTHREADS
459 void Deduplicate(void * targs) {
460   struct thread_args *args = (struct thread_args *)targs;
461   List * list = args->list;
462   Node * node;
463   Node * buffer = list->head;
464
465   #ifdef ENABLE_STATISTICS
466     stats_t* thread_stats = args->stats;
467     init_stats(thread_stats);
468   #endif //ENABLE_STATISTICS
469
470   int len = list->length;
471   for(int i = 0; i< len; i++) {
472     node = buffer;
473     buffer = buffer->next;
474     assert(node->data!=NULL);
475     //Do the processing
476     sub_Deduplicate(node->data);
477   }
478 }
479 #endif //ENABLE_PTHREADS
480
481  /* Pipeline stage function and computational kernel of
       refinement stage
482  *
483  * Actions performed:
```

```
484  *  - Take coarse chunks from fragmentation stage
485  *  - Partition data block into smaller chunks with Rabin
       rolling fingerprints
486  *  - Send resulting data chunks to deduplication stage
487  *
488  * Notes:
489  *  - Allocates mbuffers for fine-granular chunks*/
490 void FragmentRefine(void * targs) {
491   struct thread_args *args = (struct thread_args *)targs;
492   int r;
493   List * list = (List *)args->list;
494
495   chunk_t *temp;
496   chunk_t *chunk;
497   u32int * rabintab = malloc(256*sizeof rabintab[0]);
498   u32int * rabinwintab = malloc(256*sizeof rabintab[0]);
499   if(rabintab == NULL || rabinwintab == NULL) EXIT_TRACE("
       Memory allocation failed.\n");
500
501 #ifdef ENABLE_STATISTICS
502   stats_t *thread_stats = args->stats;
503   init_stats(thread_stats);
504 #endif //ENABLE_STATISTICS
505
506   int chcount = 0;
507   List * refined = emptylist();
508   Iterator * iter = init_iterator(list);
509   while (hasNext(iter)) {
510     chunk = next(iter);
511     assert(chunk!=NULL);
512     rabininit(rf_win, rabintab, rabinwintab);
513     int split;
514     chcount = 0;
515     do {
516       //Find next anchor with Rabin fingerprint
517       int offset = rabinseg(chunk->uncompressed_data.ptr,
       chunk->uncompressed_data.n, rf_win, rabintab, rabinwintab
       );
518       //Can we split the buffer?
519       if(offset < chunk->uncompressed_data.n) {
520         //Allocate a new chunk and create a new memory buffer
521         temp = (chunk_t *)malloc(sizeof(chunk_t));
522         if(temp==NULL) EXIT_TRACE("Memory allocation failed.\
       n");
523         temp->header.state = chunk->header.state;
524         temp->sequence.l1num = chunk->sequence.l1num;
```

```
525
526         //split it into two pieces
527         r = mbuffer_split(&chunk->uncompressed_data, &temp->
    uncompressed_data, offset);
528         if(r!=0) EXIT_TRACE("Unable to split memory buffer in
     refinement stage.\n");
529
530         //Set correct state and sequence numbers
531         chunk->sequence.l2num = chcount;
532         chunk->isLastL2Chunk = FALSE;
533         chcount++;
534
535         #ifdef ENABLE_STATISTICS
536           //update statistics
537           thread_stats->nChunks[CHUNK_SIZE_TO_SLOT(chunk->
    uncompressed_data.n)]++;
538         #endif //ENABLE_STATISTICS
539
540         //put it into send buffer
541         add(chunk, refined);
542         //prepare for next iteration
543         chunk = temp;
544         split = 1;
545       } else {
546         //End of buffer reached, don't split but simply
    enqueue it
547         //Set correct state and sequence numbers
548         chunk->sequence.l2num = chcount;
549         chunk->isLastL2Chunk = TRUE;
550
551         #ifdef ENABLE_STATISTICS
552           //update statistics
553           thread_stats->nChunks[CHUNK_SIZE_TO_SLOT(chunk->
    uncompressed_data.n)]++;
554         #endif //ENABLE_STATISTICS
555
556         add(chunk, refined);
557         //prepare for next iteration
558         chunk = NULL;
559         split = 0;
560       }
561     } while(split);
562   }
563
564   *(args->list_addr) = refined;
565   free(rabintab);
```

```
566   free(rabinwintab);
567   destroy_iterator(iter);
568 }
569
570 /*
571  * Pipeline stage function of fragmentation stage
572  *
573  * Actions performed:
574  *  - Read data from file (or preloading buffer)
575  *  - Perform coarse-grained chunking
576  *  - Send coarse chunks to refinement stages for further
       processing
577  *
578  * Notes:
579  * This pipeline stage is a bottleneck because it is
       inherently serial. We
580  * therefore perform only coarse chunking and pass on the
       data block as fast
581  * as possible so that there are no delays that might
       decrease scalability.
582  * With very large numbers of threads this stage will not be
       able to keep up
583  * which will eventually limit scalability. A solution to
       this is to increase
584  * the size of coarse-grained chunks with a comparable
       increase in total
585  * input size.
586  */
587 #ifdef ENABLE_PTHREADS
588 List * Fragment(void * targs){
589   struct thread_args *args = (struct thread_args *)targs;
590   size_t preloading_buffer_seek = 0;
591   int fd = args->fd;
592   int r;
593   sequence_number_t anchorcount = 0;
594
595   List * list = emptylist();
596
597   chunk_t *temp = NULL;
598   chunk_t *chunk = NULL;
599   u32int * rabintab = malloc(256*sizeof rabintab[0]);
600   u32int * rabinwintab = malloc(256*sizeof rabintab[0]);
601   if(rabintab == NULL || rabinwintab == NULL) {
602     EXIT_TRACE("Memory allocation failed.\n");
603   }
604
```

```
605   rf_win_dataprocess = 0;
606   rabininit(rf_win_dataprocess, rabintab, rabinwintab);
607
608   //Sanity check
609   if(MAXBUF < 8 * ANCHOR_JUMP) {
610     printf("WARNING: I/O buffer size is very small.
      Performance degraded.\n");
611     fflush(NULL);
612   }
613
614   //read from input file / buffer
615   while (1) {
616     size_t bytes_left; //amount of data left over in
      last_mbuffer from previous iteration
617
618     //Check how much data left over from previous iteration
      resp. create an initial chunk
619     if(temp != NULL) {
620       bytes_left = temp->uncompressed_data.n;
621     } else {
622       bytes_left = 0;
623     }
624     //Make sure that system supports new buffer size
625     if(MAXBUF+bytes_left > SSIZE_MAX) {
626       EXIT_TRACE("Input buffer size exceeds system maximum.\n
      ");
627     }
628     //Allocate a new chunk and create a new memory buffer
629     chunk = (chunk_t*)malloc(sizeof(chunk_t));
630     if(chunk==NULL) EXIT_TRACE("Memory allocation failed.\n")
      ;
631     mbuffer_create(&chunk->uncompressed_data, MAXBUF+
      bytes_left);
632
633     if(bytes_left > 0) {
634       //"Extension" of existing buffer, copy sequence number
      and left over data to beginning of new buffer
635       chunk->header.state = CHUNK_STATE_UNCOMPRESSED;
636       chunk->sequence.l1num = temp->sequence.l1num;
637       //NOTE: We cannot safely extend the current memory
      region because it has already been given to another
      thread
638       memcpy(chunk->uncompressed_data.ptr, temp->
      uncompressed_data.ptr, temp->uncompressed_data.n);
639       mbuffer_free(&temp->uncompressed_data);
640       free(temp);
```

```
641         temp = NULL;
642       } else {
643         //brand new mbuffer, increment sequence number
644         chunk->header.state = CHUNK_STATE_UNCOMPRESSED;
645         chunk->sequence.llnum = anchorcount;
646         anchorcount++;
647       }
648       //Read data until buffer full
649       size_t bytes_read=0;
650       if(conf->preloading) {
651         size_t max_read = MIN(MAXBUF, args->input_file.size-
      preloading_buffer_seek);
652         memcpy(chunk->uncompressed_data.ptr+bytes_left, args->
      input_file.buffer+preloading_buffer_seek, max_read);
653         bytes_read = max_read;
654         preloading_buffer_seek += max_read;
655       } else {
656         while(bytes_read < MAXBUF) {
657           int r = read(fd, chunk->uncompressed_data.ptr+
      bytes_left+bytes_read, MAXBUF-bytes_read);
658           if(r<0) switch(errno) {
659             case EAGAIN:
660               EXIT_TRACE("I/O error: No data available\n");
      break;
661             case EBADF:
662               EXIT_TRACE("I/O error: Invalid file descriptor\n"
      );break;
663             case EFAULT:
664               EXIT_TRACE("I/O error: Buffer out of range\n");
      break;
665             case EINTR:
666               EXIT_TRACE("I/O error: Interruption\n");break;
667             case EINVAL:
668               EXIT_TRACE("I/O error: Unable to read from file
      descriptor\n");break;
669             case EIO:
670               EXIT_TRACE("I/O error: Generic I/O error\n");
      break;
671             case EISDIR:
672               EXIT_TRACE("I/O error: Cannot read from a
      directory\n");break;
673             default:
674               EXIT_TRACE("I/O error: Unrecognized error\n");
      break;
675           }
676           if(r==0) break;
```

```
677         bytes_read += r;
678       }
679     }
680     //No data left over from last iteration and also nothing
    new read in, simply clean up and quit
681     if(bytes_left + bytes_read == 0) {
682       mbuffer_free(&chunk->uncompressed_data);
683       #ifdef ENABLE_MBUFFER_CHECK
684           m->check_flag=0;
685       #endif
686       free(chunk);
687       chunk = NULL;
688       break;
689     }
690     //Shrink buffer to actual size
691     if(bytes_left+bytes_read < chunk->uncompressed_data.n) {
692       r = mbuffer_realloc(&chunk->uncompressed_data,
    bytes_left+bytes_read);
693       assert(r == 0);
694     }
695     //Check whether any new data was read in, enqueue last
    chunk if not
696     if(bytes_read == 0) {
697       add(chunk, list);
698       //NOTE: No need to empty a full send_buf, we will break
     now and pass everything on to the queue
699       break;
700     }
701     //partition input block into large, coarse-granular
    chunks
702     int split;
703     do {
704       split = 0;
705       //Try to split the buffer at least ANCHOR_JUMP bytes
    away from its beginning
706       if(ANCHOR_JUMP < chunk->uncompressed_data.n) {
707         int offset = rabinseg(chunk->uncompressed_data.ptr +
    ANCHOR_JUMP, chunk->uncompressed_data.n - ANCHOR_JUMP,
    rf_win_dataprocess, rabintab, rabinwintab);
708         //Did we find a split location?
709         if(offset == 0) {
710           //Split found at the very beginning of the buffer (
    should never happen due to technical limitations)
711           assert(0);
712           split = 0;
```

```
713         } else if(offset + ANCHOR_JUMP < chunk->
    uncompressed_data.n) {
714           //Split found somewhere in the middle of the buffer
715           //Allocate a new chunk and create a new memory
    buffer
716           temp = (chunk_t *)malloc(sizeof(chunk_t));
717           if(temp==NULL) EXIT_TRACE("Memory allocation failed
    .\n");
718
719           int size = offset + ANCHOR_JUMP;
720
721           mbuffer_create(&temp->uncompressed_data,size);
722           memcpy(temp->uncompressed_data.ptr, chunk->
    uncompressed_data.ptr ,size);
723
724           //split it into two pieces
725           void * p = chunk->uncompressed_data.ptr + size;
726           int p_n = chunk->uncompressed_data.n - size;
727           mcb_t * p_mcb = chunk->uncompressed_data.mcb;
728
729           chunk->uncompressed_data.ptr = temp->
    uncompressed_data.ptr;
730           chunk->uncompressed_data.n = size;
731           chunk->uncompressed_data.mcb = temp->
    uncompressed_data.mcb;
732
733           temp->uncompressed_data.ptr = p;
734           temp->uncompressed_data.n = p_n;
735           temp->uncompressed_data.mcb = p_mcb;
736
737           #ifdef ENABLE_MBUFFER_CHECK
738             m2->check_flag=MBUFFER_CHECK_MAGIC;
739           #endif
740
741           temp->header.state = CHUNK_STATE_UNCOMPRESSED;
742           temp->sequence.l1num = anchorcount;
743           anchorcount++;
744
745           //put it into send buffer
746           add(chunk, list);
747           //prepare for next iteration
748           chunk = temp;
749           temp = NULL;
750           split = 1;
751         } else {
```

```
752             //Due to technical limitations we can't distinguish
    the cases "no split" and "split at end of buffer"
753             //This will result in some unnecessary (and
    unlikely) work but yields the correct result eventually.
754             temp = chunk;
755             chunk = NULL;
756             split = 0;
757           }
758         } else {
759           //NOTE: We don't process the stub, instead we try to
    read in more data so we might be able to find a proper
    split.
760           //      Only once the end of the file is reached do
    we get a genuine stub which will be enqueued right after
    the read operation.
761           temp = chunk;
762           chunk = NULL;
763           split = 0;
764         }
765       } while(split);
766     }
767     free(rabintab);
768     free(rabinwintab);
769     return list;
770   }
771   #endif //ENABLE_PTHREADS
772
773   //Write the compressed data to the output file.
774   #ifdef ENABLE_PTHREADS
775   void Write(Compressed_data ** data, int * counts) {
776     int fd = 0;
777     fd = create_output_file(conf->outfile);
778
779     for (int i = 0; i<conf->nthreads; i++){
780       for (int j = 0; j < counts[i];j++){
781         xwrite(fd, data[i][j].data, data[i][j].size);
782         free(data[i][j].data);
783       }
784       free(data[i]);
785     }
786     close(fd);
787   }
788   #endif //ENABLE_PTHREADS
789
```

```
790 /*
        ------------------------------------------------------------------------
        */
791 /* Encode
792  * Compress an input stream
793  *
794  * Arguments:
795  *   conf:    Configuration parameters
796  *
797  */
798 void Encode(config_t * _conf) {
799   printf("**Dedup encoding with minimal virtualization
        overhead**\n");
800   struct stat filestat;
801   int32 fd;
802
803   conf = _conf;
804
805 #ifdef ENABLE_STATISTICS
806   init_stats(&stats);
807 #endif
808
809   //Create chunk cache
810   cache = hashtable_create(65536, hash_from_key_fn,
        keys_equal_fn, FALSE);
811   if(cache == NULL) {
812     printf("ERROR: Out of memory\n");
813     exit(1);
814   }
815
816 #ifdef ENABLE_PTHREADS
817   struct thread_args data_process_args;
818 #else
819   struct thread_args generic_args;
820 #endif //ENABLE_PTHREADS
821
822   /* src file stat */
823   if (stat(conf->infile, &filestat) < 0)
824       EXIT_TRACE("stat() %s failed: %s\n", conf->infile,
        strerror(errno));
825
826   if (!S_ISREG(filestat.st_mode))
827     EXIT_TRACE("not a normal file: %s\n", conf->infile);
828 #ifdef ENABLE_STATISTICS
829   stats.total_input = filestat.st_size;
830 #endif //ENABLE_STATISTICS
```

```
831
832   /* src file open */
833   if((fd = open(conf->infile, O_RDONLY | O_LARGEFILE)) < 0)
834     EXIT_TRACE("%s file open error %s\n", conf->infile,
      strerror(errno));
835
836   //Load entire file into memory if requested by user
837   void *preloading_buffer = NULL;
838   if(conf->preloading) {
839     size_t bytes_read=0;
840     int r;
841
842     preloading_buffer = malloc(filestat.st_size);
843     if(preloading_buffer == NULL)
844       EXIT_TRACE("Error allocating memory for input buffer.\n
      ");
845
846     //Read data until buffer full
847     while(bytes_read < filestat.st_size) {
848       r = read(fd, preloading_buffer+bytes_read, filestat.
      st_size-bytes_read);
849       if(r<0) switch(errno) {
850         case EAGAIN:
851           EXIT_TRACE("I/O error: No data available\n");break;
852         case EBADF:
853           EXIT_TRACE("I/O error: Invalid file descriptor\n");
      break;
854         case EFAULT:
855           EXIT_TRACE("I/O error: Buffer out of range\n");
      break;
856         case EINTR:
857           EXIT_TRACE("I/O error: Interruption\n");break;
858         case EINVAL:
859           EXIT_TRACE("I/O error: Unable to read from file
      descriptor\n");break;
860         case EIO:
861           EXIT_TRACE("I/O error: Generic I/O error\n");break;
862         case EISDIR:
863           EXIT_TRACE("I/O error: Cannot read from a directory
      \n");break;
864         default:
865           EXIT_TRACE("I/O error: Unrecognized error\n");break
      ;
866       }
867       if(r==0) break;
868       bytes_read += r;
```

```
869     }
870 #ifdef ENABLE_PTHREADS
871     data_process_args.input_file.size = filestat.st_size;
872     data_process_args.input_file.buffer = preloading_buffer;
873 #else
874     generic_args.input_file.size = filestat.st_size;
875     generic_args.input_file.buffer = preloading_buffer;
876 #endif //ENABLE_PTHREADS
877   }
878
879   data_process_args.tid = 0;
880   data_process_args.fd = fd;
881
882 #ifdef ENABLE_PARSEC_HOOKS
883     __parsec_roi_begin();
884 #endif
885
886   int threadCount = conf->nthreads;
887   threadpool pool = thpool_init(threadCount);
888
889   List * fragmented = Fragment(&data_process_args);
890   //clean up after preloading
891     if(conf->preloading) free(preloading_buffer);
892
893
894   List ** refined = split(threadCount, fragmented);
895   int i = 0;
896   struct thread_args anchor_thread_args[threadCount];
897   stats_t threads_anchor_rv[threadCount];
898   for (i = 0; i < threadCount; i ++) {
899      anchor_thread_args[i].tid = i;
900      anchor_thread_args[i].list = refined[i];
901
902      anchor_thread_args[i].stats = &threads_anchor_rv[i];
903      anchor_thread_args[i].list_addr = &(refined[i]);
904
905      thpool_add_work(pool, FragmentRefine, &
      anchor_thread_args[i]);
906   }
907   thpool_wait(pool);
908
909   List * refined_merged = emptylist();
910   for (i = 0; i < threadCount; i++) refined_merged = merge(
      refined_merged,refined[i]);
911   List ** dedup = split_mod(threadCount, refined_merged);
912
```

```
913   struct thread_args chunk_thread_args[threadCount];
914   stats_t threads_chunk_rv[threadCount];
915   for (i = 0; i < threadCount; i ++) {
916     chunk_thread_args[i].tid = i;
917     chunk_thread_args[i].list = dedup[i];
918     chunk_thread_args[i].stats = &threads_chunk_rv[i];
919     thpool_add_work(pool, Deduplicate, &chunk_thread_args[i])
      ;
920   }
921   thpool_wait(pool);
922
923   List ** compress = zip_split(threadCount, dedup);
924   Compressed_data * total_compressed_data[threadCount];
925   int buffer_counts[threadCount];
926
927   struct thread_args compress_thread_args[threadCount];
928   stats_t threads_compress_rv[threadCount];
929   for (i = 0; i < threadCount; i ++) {
930     compress_thread_args[i].tid = i;
931     compress_thread_args[i].list = compress[i];
932     int write_buffer_count = compress[i]->length/1000 + ((
      compress[i]->length % 1000 == 0) ? 0:1);
933     total_compressed_data[i] = malloc(write_buffer_count *
      sizeof(Compressed_data));
934     buffer_counts[i] = write_buffer_count;
935     compress_thread_args[i].compressed_data =
      total_compressed_data[i];
936     compress_thread_args[i].stats = &threads_compress_rv[i];
937     thpool_add_work(pool, Compress, &compress_thread_args[i])
      ;
938   }
939   thpool_wait(pool);
940   thpool_destroy(pool);
941
942   Write(total_compressed_data, buffer_counts);
943
944 #ifdef ENABLE_PARSEC_HOOKS
945   __parsec_roi_end();
946 #endif
947
948 #ifdef ENABLE_STATISTICS
949   //Merge everything into global 'stats' structure
950   for(i=0; i<conf->nthreads; i++) merge_stats(&stats, &
      threads_anchor_rv[i]);
951   for(i=0; i<conf->nthreads; i++) merge_stats(&stats, &
      threads_chunk_rv[i]);
```

```
952   for(i=0; i<conf->nthreads; i++) merge_stats(&stats, &
        threads_compress_rv[i]);
953 #endif //ENABLE_STATISTICS
954
955   // clean up with the src file
956   if (conf->infile != NULL) close(fd);
957
958   hashtable_destroy(cache, FALSE);
959
960 #ifdef ENABLE_STATISTICS
961    //dest file stat
962   if (stat(conf->outfile, &filestat) < 0)
963       EXIT_TRACE("stat() %s failed: %s\n", conf->outfile,
        strerror(errno));
964   stats.total_output = filestat.st_size;
965
966   //Analyze and print statistics
967   if(conf->verbose) print_stats(&stats);
968 #endif //ENABLE_STATISTICS
969
970 }
```

Bibliography

[1] P. Neto. “Demystifying cloud computing”. In: *Proceeding of doctoral symposium on informatics engineering*. Vol. 24. Citeseer. 2011, pp. 16–21.

[2] Eurostat. *Cloud computing - statistics on the use by enterprises*. Jan. 2021. URL: https://ec.europa.eu/eurostat/statistics-explained/index.php/Cloud_computing_-_statistics_on_the_use_by_enterprises#Use_of_cloud_computing:_highlights (visited on 04/29/2021).

[3] N. Taleb and E. A. Mohamed. “Cloud computing trends: A literature review”. In: *Academic Journal of Interdisciplinary Studies* 9.1 (2020), pp. 91–91.

[4] Q. Zhang, L. Cheng, and R. Boutaba. “Cloud computing: state-of-the-art and research challenges”. In: *Journal of internet services and applications* 1.1 (2010), pp. 7–18.

[5] L. Malhotra, D. Agarwal, A. Jaiswal, et al. “Virtualization in cloud computing”. In: *J. Inform. Tech. Softw. Eng* 4.2 (2014), pp. 1–3.

[6] G. J. Popek and R. P. Goldberg. “Formal requirements for virtualizable third generation architectures”. In: *Communications of the ACM* 17.7 (1974), pp. 412–421.

[7] G. P. C. Tran, Y.-A. Chen, D.-I. Kang, J. P. Walters, and S. P. Crago. “Hypervisor performance analysis for real-time workloads”. In: *2016 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE. 2016, pp. 1–7.

[8] N. Khanghahi and R. Ravanmehr. “Cloud computing performance evaluation: issues and challenges”. In: *Comput* 5.1 (2013), pp. 29–41.

[9] G. Lettieri, V. Maffione, and L. Rizzo. “A study of I/O performance of virtual machines”. In: *The Computer Journal* 61.6 (2018), pp. 808–831.

[10] X. Ding and J. Shan. “Diagnosing Virtualization Overhead for Multi-threaded Computation on Multicore Platforms”. In: *CloudCom’15*. 2015, pp. 226–233.

[11] R. Scroggins. “Virtualization technology literature review”. In: *Global Journal of Computer Science and Technology* (2013).

[12] J. Li, S. Xue, W. Zhang, Z. Qi, et al. “When I/O interrupt becomes system bottleneck: Efficiency and scalability enhancement for SR-IOV network virtualization”. In: *IEEE TCC* 7.4 (2019).

[13] J. Shan, X. Ding, and N. Gehani. “APPLES: Efficiently handling spin-lock synchronization on virtualized platforms”. In: *IEEE Transactions on Parallel and Distributed Systems* 28.7 (2016), pp. 1811–1824.

[14] X. Ding, P. B. Gibbons, M. A. Kozuch, and J. Shan. “Gleaner: Mitigating the Blocked-Waiter Wakeup Problem for Virtualized Multicore Applications”. In: *USENIX ATC 2014*. 2014, pp. 73–84.

[15] The KVM halt polling system. URL: `https://www.kernel.org/doc/Documentation/virtual/kvm/halt-polling.txt` (visited on 07/07/2021).

[16] J. Nakajima. *Reviewing Unused and New Features for Interrupt/APIC Virtualization*. 2012.

[17] P. E. Kampert. “A taxonomy of virtualization technologies”. In: (2010).

[18] M. Malawski, A. Gajek, A. Zima, B. Balis, and K. Figiela. “Serverless execution of scientific workflows: Experiments with hyperflow, aws lambda and google cloud functions”. In: *Future Generation Computer Systems* (2017).

[19] A. Iosup, S. Ostermann, M. N. Yigitbasi, R. Prodan, T. Fahringer, and D. Epema. “Performance analysis of cloud computing services for many-tasks scientific computing”. In: *IEEE TPDS* 22.6 (2011), pp. 931–945.

[20] R. R. Expósito, G. L. Taboada, S. Ramos, J. Touriño, and R. Doallo. “Performance analysis of HPC applications in the cloud”. In: *Future Generation Computer Systems* 29.1 (2013), pp. 218–229.

[21] H. N. Palit, X. Li, S. Lu, L. C. Larsen, and J. A. Setia. “Evaluating hardware-assisted virtualization for deploying HPC-as-a-service”. In: *Proceedings of the 7th international workshop on Virtualization technologies in distributed computing*. ACM. 2013, pp. 11–20.

[22] S. Benedict. “Performance issues and performance analysis tools for HPC cloud applications: a survey”. In: *Computing* 95.2 (2013), pp. 89–108.

[23] L. Bo, Z. Zhenliu, and W. Xiangfeng. “A survey of HPC Development”. In: *2012 International Conference on Computer Science and Electronics Engineering*. Vol. 2. IEEE. 2012, pp. 103–106.

[24] D. M. Tullsen, S. J. Eggers, and H. M. Levy. "Simultaneous multithreading: Maximizing on-chip parallelism". In: *Proceedings of the 22nd annual international symposium on Computer architecture*. 1995, pp. 392–403.

[25] T. Alsop. *Share of the global server processor market by type from 2018 to 2019*. Apr. 2021. URL: https://www.statista.com/statistics/915080/global-market-share-held-by-server-vendors/ (visited on 05/05/2021).

[26] D. Marshall. "Understanding full virtualization, paravirtualization, and hardware assist". In: *VMWare White Paper* 17 (2007), p. 725.

[27] J. Fisher-Ogden. "Hardware support for efficient virtualization". In: *University of California, San Diego, Tech. Rep* 12 (2006).

[28] T. Alsop. *Share of the global server market in the first half of 2018 and 2019, by virtualization type*. May 2020. URL: https://www.statista.com/statistics/915091/global-server-share-physical-virtual/ (visited on 05/05/2021).

[29] S. Schildermans, J. Shan, K. Aerts, J. Jackrel, and X. Ding. "Virtualization Overhead of Multithreading in X86 State-of-the-Art & Remaining Challenges". In: *IEEE Transactions on Parallel and Distributed Systems* 32.10 (2021), pp. 2557–2570.

[30] S. Schildermans, K. Aerts, J. Shan, and X. Ding. "Paratick: Reducing Timer Overhead in Virtual Machines". In: *50th International Conference on Parallel Processing*. 2021, pp. 1–10.

[31] S. Schildermans, K. Aerts, J. Shan, and X. Ding. "Ptlbmalloc2: Reducing TLB Shootdowns with High Memory Efficiency". In: *ISPA-BDCloud-SocialCom-SustainCom 2020* (2020), pp. 76–83.

[32] S. Schildermans and K. Aerts. "Towards High-Level Software Approaches to Reduce Virtualization Overhead for Parallel Applications". In: *2018 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE. 2018, pp. 193–197.

[33] S. N. T.-c. Chiueh and S. Brook. "A survey on virtualization technologies". In: *Rpe Report* 142 (2005).

[34] J. E. Smith and R. Nair. "The architecture of virtual machines". In: *Computer* 38.5 (2005), pp. 32–38.

[35] R. P. Goldberg. "Survey of virtual machine research". In: *Computer* 7.6 (1974), pp. 34–45.

[36] NI. *Introduction to the NI Real-Time Hypervisor*. 2009.

[37] H. Lee. "Virtualization basics: Understanding techniques and fundamentals". In: *School of Informatics and Computing Indiana University 815 E 10th St. Bloomington IN 47408*. 2014.

[38] S. Alliance. “Virtualization: State of the Art”. In: (2008). URL: http://scopealliance.org/sites/default/files/documents/SCOPE-Virtualization-StateofTheArt-Version-1.0.pdf.

[39] *Chapter 10. Technical background*. URL: https://www.virtualbox.org/manual/ch10.html (visited on 05/12/2021).

[40] Y. Goto. “Kernel-based virtual machine technology”. In: *Fujitsu Scientific and Technical Journal* 47.3 (2011), pp. 362–368.

[41] T. Maeda and A. Yonezawa. “Kernel Mode Linux: Toward an operating system protected by a type theory”. In: *Annual Asian Computing Science Conference*. Springer. 2003, pp. 3–17.

[42] *Intel 64 and IA-32 Architectures Software Developer’s Manual*. Intel. Santa Clara, CA, USA, May 2019. URL: https://software.intel.com/content/www/us/en/develop/articles/intel-sdm.html.

[43] N. Penneman, D. Kudinskas, A. Rawsthorne, B. De Sutter, and K. De Bosschere. “Formal virtualization requirements for the ARM architecture”. In: *Journal of Systems Architecture* 59.3 (2013), pp. 144–154.

[44] J. White and A. Pilbeam. “A survey of virtualization technologies with performance testing”. In: *arXiv preprint arXiv:1010.3233* (2010).

[45] J. Shuja, A. Gani, A. Naveed, E. Ahmed, and C.-H. Hsu. “Case of ARM emulation optimization for offloading mechanisms in mobile cloud computing”. In: *Future Generation Computer Systems* 76 (2017), pp. 407–417.

[46] K. Adams and O. Agesen. “A comparison of software and hardware techniques for x86 virtualization”. In: *ACM Sigplan Notices* 41.11 (2006), pp. 2–13.

[47] M. Probst. “Dynamic binary translation”. In: *UKUUG Linux Developer’s Conference*. Vol. 2002. 2002.

[48] R. Community. *Nested Virtualization With Binary Translation: Back to the Future*. Nov. 2013. URL: https://blogs.oracle.com/ravello/nested-virtualization-with-binary-translation (visited on 06/02/2021).

[49] M. Rosenblum and T. Garfinkel. “Virtual machine monitors: Current technology and future trends”. In: *Computer* 38.5 (2005), pp. 39–47.

[50] H. Kim, S. Kim, J. Jeong, J. Lee, and S. Maeng. "Demand-based Coordinated Scheduling for SMP VMs". In: *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*. Houston, Texas, USA, 2013, pp. 369–380. ISBN: 978-1-4503-1870-9. DOI: 10.1145/2451116.2451156. URL: http://doi.acm.org/10.1145/2451116.2451156.

[51] A. Menon, J. R. Santos, Y. Turner, G. Janakiraman, and W. Zwaenepoel. "Diagnosing performance overheads in the xen virtual machine environment". In: *Proceedings of the 1st ACM/USENIX international conference on Virtual execution environments*. 2005, pp. 13–23.

[52] VMWare. *Hardware-Assisted Memory virtualization*. Apr. 2018. URL: https://docs.vmware.com/en/VMware-vSphere/6.5/com.vmware.vsphere.resmgmt.doc/GUID-69CDC049-8B42-4D26-8B47-94961B1777A4.html (visited on 06/07/2021).

[53] X. Wang, J. Zang, Z. Wang, Y. Luo, and X. Li. "Selective hardware/software memory virtualization". In: *ACM SIGPLAN Notices* 46.7 (2011), pp. 217–226.

[54] E. Alkassar, E. Cohen, M. Hillebrand, M. Kovalev, and W. J. Paul. "Verifying shadow page table algorithms". In: *Formal Methods in Computer Aided Design*. 2010, pp. 267–270.

[55] J. Gandhi, M. D. Hill, and M. M. Swift. "Agile paging: Exceeding the best of nested and shadow paging". In: *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*. IEEE. 2016, pp. 707–718.

[56] B. T. Djomgwe, P. Yuhala, A. Tchana, F. Hermenier, D. Hagimont, and G. Muller. "(No) Compromis: Paging Virtualization Is Not a Fatality". In: *VEE 2021-17th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. 2021, pp. 1–12.

[57] A. Krapf. "XEN Memory Management (Intel IA-32)". In: *INRIA Sophia Antipolis-Méditerranée Research Centre* (2007).

[58] C. Waldspurger and M. Rosenblum. "I/o virtualization". In: *Communications of the ACM* 55.1 (2012), pp. 66–73.

[59] Y. Luo. "Network I/O virtualization for cloud computing". In: *IT professional* 12.5 (2010), pp. 36–41.

[60] Y. Dong, J. Dai, Z. Huang, H. Guan, K. Tian, and Y. Jiang. "Towards high-quality I/O virtualization". In: *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*. 2009, pp. 1–8.

[61] M. Jones. *Linux virtualization and PCI passthrough*. Oct. 2009. URL: https://developer.ibm.com/tutorials/l-pci-passthrough/ (visited on 06/08/2021).

[62] B. Zhang, X. Wang, R. Lai, L. Yang, Y. Luo, X. Li, and Z. Wang. "A survey on i/o virtualization and optimization". In: *2010 Fifth Annual ChinaGrid Conference*. IEEE. 2010, pp. 117–123.

[63] F. Bellard. "QEMU, a fast and portable dynamic translator." In: *USENIX annual technical conference, FREENIX Track*. Vol. 41. Califor-nia, USA. 2005, p. 46.

[64] J. R. Santos, Y. Turner, G. J. Janakiraman, and I. Pratt. "Bridging the Gap between Software and Hardware Techniques for I/O Virtualization." In: *USENIX Annual Technical Conference*. 2008, pp. 29–42.

[65] *Intel Virtualization Technology for Directed I/O*. Intel. Santa Clara, CA, USA, Apr. 2021. URL: https://software.intel.com/content/www/us/en/develop/download/intel-virtualization-technology-for-directed-io-architecture-specification.html.

[66] PCI-SIG. *Single Root I/O Virtualiization and Sharing Specificatiion Revision 1.1*. Jan. 2010. URL: https://composter.com.ua/documents/sr-iov1_1_20Jan10_cb.pdf (visited on 06/09/2021).

[67] Y. Dong, X. Yang, J. Li, G. Liao, K. Tian, and H. Guan. "High performance network virtualization with SR-IOV". In: *Journal of Parallel and Distributed Computing* 72.11 (2012), pp. 1471–1480.

[68] V. G. da Silva, M. Kirikova, and G. Alksnis. "Containers for virtualization: An overview". In: *Applied Computer Systems* 23.1 (2018), pp. 21–27.

[69] J. Frazelle. *Setting the Record Straight: containers vs. Zones vs. Jails vs. VMs*. Mar. 2017. URL: https://blog.jessfraz.com/post/containers-zones-jails-vms/ (visited on 06/18/2021).

[70] R. Morabito, J. Kjällman, and M. Komu. "Hypervisors vs. lightweight virtualization: a performance comparison". In: *2015 IEEE International Conference on Cloud Engineering*. IEEE. 2015, pp. 386–393.

[71] D. Bernstein. "Containers and cloud: From lxc to docker to kubernetes". In: *IEEE Cloud Computing* 1.3 (2014), pp. 81–84.

[72] T. Bui. "Analysis of docker security". In: *arXiv preprint arXiv:1501.02967* (2015).

[73] Canonical. *Infrastructure for container projects*. URL: https://linuxcontainers.org/ (visited on 06/18/2021).

[74] D. Drewanz and L. Grimmer. *The Role of Oracle Solaris Zones and Linux Containers in a Virtualization Strategy*. Jan. 2013. URL: https://www.oracle.com/technical-resources/articles/it-infrastructure/admin-zones-containers-virtualization.html (visited on 06/18/2021).

[75] C. Tozzi. *Jails, LXC and Beyond: Container Platform Round-Up*. July 2017. URL: https://containerjournal.com/features/jails-lxc-beyond-container-platform-round/ (visited on 06/18/2021).

[76] Docker. *Docker overview*. URL: https://docs.docker.com/get-started/overview/ (visited on 06/18/2021).

[77] R. Singh. *LXD vs Docker*. 2017. URL: https://linuxhint.com/lxd-vs-docker/ (visited on 06/18/2021).

[78] *History of Operating Systems*. URL: https://sites.google.com/site/optsytms/history-of-operating-systems (visited on 06/23/2021).

[79] A. S. Tanenbaum and H. Bos. *Modern operating systems*. Pearson, 2015.

[80] O. Ike-Nwosu. *Inside the Python Virtual Machine*. 2015.

[81] G. Kumar. *Understanding the difference between JDK, JRE and JVM is important in Java*. Oct. 2015. URL: https://www.linkedin.com/pulse/understanding-difference-between-jdk-jre-jvm-important-kumar (visited on 06/24/2021).

[82] J. Thönes. "Microservices". In: *IEEE software* 32.1 (2015), pp. 116–116.

[83] G. McGrath and P. R. Brenner. "Serverless computing: Design, implementation, and performance". In: *ICDCSW'17*. IEEE. 2017, pp. 405–410.

[84] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S. Smith, S. Hand, and J. Crowcroft. "Unikernels: Library operating systems for the cloud". In: *ACM SIGARCH Computer Architecture News* 41.1 (2013), pp. 461–472.

[85] A. Kivity, D. Laor, G. Costa, P. Enberg, N. Har'El, D. Marti, and V. Zolotarov. "OSv—optimizing the operating system for virtual machines". In: *2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14)*. 2014, pp. 61–72.

[86] I. Briggs, M. Day, Y. Guo, P. Marheine, and E. Eide. "A performance evaluation of unikernels". In: *Technical Report*. 2014.

[87] R. Morabito, J. Kjällman, and M. Komu. "Hypervisors vs. lightweight virtualization: a performance comparison". In: *2015 IEEE International Conference on Cloud Engineering*. IEEE. 2015, pp. 386–393.

[88] A. Raza, P. Sohal, J. Cadden, J. Appavoo, U. Drepper, R. Jones, O. Krieger, R. Mancuso, and L. Woodman. "Unikernels: The next stage of linux's dominance". In: *Proceedings of the Workshop on Hot Topics in Operating Systems*. 2019, pp. 7–13.

[89] D. Williams and R. Koller. "Unikernel monitors: extending minimalism outside of the box". In: *8th {USENIX} Workshop on Hot Topics in Cloud Computing (HotCloud 16)*. 2016.

[90] *VNC (Virtual Network Computing)*. URL: `https://www.raspberrypi.org/documentation/remote-access/vnc/` (visited on 06/24/2021).

[91] *Data Storage - Logical Block Addressing (LBA)*. URL: `https://datacadamia.com/io/drive/lba` (visited on 07/05/2021).

[92] S. Lee, M. Liu, S. Jun, S. Xu, J. Kim, et al. "Application-managed flash". In: *14th {USENIX} Conference on File and Storage Technologies ({FAST} 16)*. 2016, pp. 339–353.

[93] C. Hoffman. *Beginner Geek: Hard Disk Partitions Explained*. July 2017. URL: `https://www.howtogeek.com/184659/beginner-geek-hard-disk-partitions-explained/` (visited on 07/05/2021).

[94] P. Gupta and C. S. S. Asia. "Storage Virtualization: What, Why, Where and How". In: *The Storage Networking Industry Association (SNIA)* (2008).

[95] G. Smida. *DAS RAID NAS SAN*. Dec. 2012. URL: `https://www.slideshare.net/gsmida/das-raid-nas-san`.

[96] P. Raj and A. Raman. "Software-defined storage (SDS) for storage virtualization". In: *Software-defined cloud centers*. Springer, 2018, pp. 35–64.

[97] A. Gillis. *RAID (redundant array of independent disks)*. Feb. 2020. URL: `https://searchstorage.techtarget.com/definition/RAID` (visited on 06/30/2021).

[98] VMWare. *Understanding the DNA of Software Defined Storage*. URL: `https://www.vmware.com/content/dam/digitalmarketing/vmware/en/pdf/solutions/understanding-the-dna-of-software-defined-storage-tech-trends.pdf`.

[99] D. Kreutz, F. M. Ramos, P. E. Verissimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig. "Software-defined networking: A comprehensive survey". In: *Proceedings of the IEEE* 103.1 (2014), pp. 14–76.

[100] IBM. *Virtual IP address*. 2020. URL: `https://www.ibm.com/docs/en/aix/7.2?topic=protocol-virtual-ip-address` (visited on 07/05/2021).

[101] A. Mumford. *What's the difference between a LAN and a WAN?* July 2019. URL: https://purple.ai/blogs/whats-the-difference-between-a-lan-and-a-wan/ (visited on 07/05/2021).

[102] M. Heller. *What you need to know about VPN technologies*. Aug. 2006. URL: https://www.computerworld.com/article/2546283/what-you-need-to-know-about-vpn-technologies.html (visited on 07/05/2021).

[103] L. Liu, H. Wang, A. Wang, M. Xiao, Y. Cheng, and S. Chen. "vCPU As a Container: Towards Accurate CPU Allocation for VMs". In: *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. Providence, RI, USA: ACM, 2019, pp. 193–206. ISBN: 978-1-4503-6020-3. DOI: 10.1145/3313808.3313814. URL: http://doi.acm.org/10.1145/3313808.3313814.

[104] D. Gupta, S. Lee, M. Vrable, S. Savage, A. C. Snoeren, G. Varghese, G. M. Voelker, and A. Vahdat. "Difference Engine: Harnessing Memory Redundancy in Virtual Machines". In: *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*. 2008, pp. 309–322.

[105] *Virtio: Paravirtualized drivers for KVM/Linux*. URL: https://www.linux-kvm.org/page/Virtio (visited on 07/06/2021).

[106] S. W. Devine, L. S. Rogel, P. P. Bungale, et al. *Virtualization with shadow page tables*. US Patent 8,464,022. June 2013.

[107] R. Bhargava, B. Serebrin, F. Spadini, and S. Manne. "Accelerating two-dimensional page walks for virtualized systems". In: *SIGOPS Oper. Syst. Rev.* 42.2 (2008), pp. 26–35.

[108] J. Rao, K. Wang, X. Zhou, and C.-Z. Xu. "Optimizing virtual machine scheduling in NUMA multicore systems". In: *HPCA'13*. IEEE. 2013, pp. 306–317.

[109] B. Teabe, V. Nitu, A. Tchana, and D. Hagimont. "The Lock Holder and the Lock Waiter Pre-emption Problems: Nip Them in the Bud Using Informed Spinlocks (I-Spinlock)". In: *EuroSys '17*. 2017, pp. 286–297.

[110] T. Friebel and S. Biemueller. "How to deal with lock holder preemption". In: *Xen Summit North America* (2008).

[111] J. M. Mellor-Crummey and M. L. Scott. "Algorithms for scalable synchronization on shared-memory multiprocessors". In: *ACM Transactions on Computer Systems (TOCS)* 9.1 (1991), pp. 21–65.

[112] AMD. *AMD64 Architecture Programmer's Manual: Volumes 1-5*. Nov. 2020. URL: https://www.amd.com/system/files/TechDocs/40332.pdf.

[113] K. Raghavendra. *Paravirtualized ticket spinlocks*. June 2013. URL: https://lwn.net/Articles/556141/ (visited on 11/10/2021).

[114] J. Ouyang and J. R. Lange. "Preemptable Ticket Spinlocks: Improving Consolidated Performance in the Cloud". In: *VEE'13*. Houston, Texas, USA, 2013, pp. 191–200. ISBN: 978-1-4503-1266-0. DOI: 10.1145/2451512.2451549. URL: http://doi.acm.org/10.1145/2451512.2451549.

[115] Torvalds. *torvalds/linux*. July 2021. URL: https://github.com/torvalds/linux (visited on 07/07/2021).

[116] C. Villavieja, V. Karakostas, L. Vilanova, Y. Etsion, A. Ramirez, A. Mendelson, N. Navarro, A. Cristal, and O. S. Unsal. "Didi: Mitigating the performance impact of TLB shootdowns using a shared TLB directory". In: *PACT 2011*, pp. 340–349.

[117] J. Ouyang, J. R. Lange, and H. Zheng. "Shoot4U: Using VMM assists to optimize TLB operations on preempted vCPUs". In: *VEE'16* (2016).

[118] M. Liu and T. Li. "Optimizing virtual machine consolidation performance on NUMA server architecture for cloud workloads". In: *2013 IEEE 19th International Symposium on High Performance Computer Architecture (HPCA)*. IEEE. 2014, pp. 325–336.

[119] *Domain XML format*. URL: https://libvirt.org/formatdomain.html (visited on 07/07/2021).

[120] AMD. *Leadership High Performance Computing*. AMD. June 5, 2020. URL: https://ir.amd.com/static-files/fd06c15e-0241-424d-9fd9-5a469d96012d (visited on 07/21/2020).

[121] The Red Hat Enterprise Linux Team. *Red Hat: Leading the enterprise Linux server market*. Dec. 2019. URL: https://www.redhat.com/en/blog/red-hat-leading-enterprise-linux-server-market (visited on 07/23/2020).

[122] VMware. *Host Power Management in VMware vSphere 5.5*. URL: http://www.vmware.com/resources/techresources/10205.

[123] Distrowatch. *DistroWatch Project Ranking*. 2020. URL: https://distrowatch.com/dwres.php?resource=ranking&sort=votes (visited on 07/23/2020).

[124] X. Zhan, Y. Bao, C. Bienia, and K. Li. "PARSEC3.0: A Multicore Benchmark Suite with Network Stacks and SPLASH-2X". In: *SIGARCH Comput. Archit. News* 44.5 (Feb. 2017), pp. 1–16. ISSN: 0163-5964. DOI: 10.1145/3053277.3053279. URL: https://doi.org/10.1145/3053277.3053279.

[125] *Open-Source, Automated Benchmarking*. 2021. URL: `https://www.phoronix-test-suite.com/` (visited on 04/24/2021).

[126] R. Feldt and A. Magazinius. "Validity threats in empirical software engineering research-an initial survey." In: *Seke*. 2010, pp. 374–379.

[127] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in software engineering*. Springer Science & Business Media, 2012.

[128] J. Li, Q. Wang, D. Jayasinghe, J. Park, T. Zhu, and C. Pu. "Performance overhead among three hypervisors: An experimental study using hadoop benchmarks". In: *IEEE BigData Congress'13*. 2013, pp. 9–16.

[129] P. Luszczek, E. Meek, S. Moore, D. Terpstra, V. M. Weaver, and J. Dongarra. "Evaluation of the HPC challenge benchmarks in virtualized environments". In: *Proceedings of the 2011 international conference on Parallel Processing - Volume 2*. Springer-Verlag, 2011, pp. 436–445.

[130] U. F. Minhas, J. Yadav, A. Aboulnaga, and K. Salem. "Database systems on virtual machines: How much do you lose?" In: *2008 IEEE 24th International Conference on Data Engineering Workshop*. 2008, pp. 35–41.

[131] A. J. Younge, R. Henschel, J. T. Brown, G. Von Laszewski, J. Qiu, and G. C. Fox. "Analysis of virtualization technologies for high performance computing environments". In: *IEEE CLOUD'11*. 2011, pp. 9–16.

[132] J. P. Walters, V. Chaudhary, M. Cha, S. Guercio Jr, and S. Gallo. "A comparison of virtualization technologies for HPC". In: *Advanced Information Networking and Applications, 2008. AINA 2008. 22nd International Conference on*. IEEE. 2008, pp. 861–868.

[133] J. Han, J. Ahn, C. Kim, Y. Kwon, Y.-r. Choi, and J. Huh. "The effect of multi-core on HPC applications in virtualized systems". In: *Euro-Par 2010 Parallel Processing Workshops*. Springer. 2011, pp. 615–623.

[134] M. H. Jamal, A. Qadeer, W. Mahmood, A. Waheed, and J. J. Ding. "Virtual machine scalability on multi-core processors based servers for cloud computing workloads". In: *Networking, Architecture, and Storage, 2009. NAS 2009. IEEE International Conference on*. IEEE. 2009, pp. 90–97.

[135] E. Walker. "Benchmarking amazon EC2 for high-performance scientific computing". In: *; login:: the magazine of USENIX & SAGE* 33.5 (2008), pp. 18–23.

[136] X. Song, H. Chen, and B. Zang. *Characterizing the Performance and Scalability of Many-core Applications on Virtualized Platforms*. Tech. rep. FDUPPITR-2010-002. Parallel Processing Institute, Fudan University, 2010.

[137] P. Padala, X. Zhu, Z. Wang, S. Singhal, K. G. Shin, et al. “Performance evaluation of virtualization technologies for server consolidation”. In: *HP Labs Tec. Report* 137 (2007).

[138] Y. Zhao, J. Rao, and Q. Yi. “Characterizing and optimizing the performance of multithreaded programs under interference”. In: *PACT 2016*. Sept. 2016, pp. 287–297.

[139] Y. Koh, R. Knauerhase, P. Brett, M. Bowman, Z. Wen, and C. Pu. “An analysis of performance interference effects in virtual environments”. In: *2007 IEEE International Symposium on Performance Analysis of Systems & Software*. 2007, pp. 200–209.

[140] J. Liu. “Evaluating standard-based self-virtualizing devices: A performance study on 10 GbE NICs with SR-IOV support”. In: *IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*. IEEE. 2010, pp. 1–12.

[141] L. Youseff, K. Seymour, H. You, D. Zagorodnov, J. Dongarra, and R. Wolski. “Paravirtualization effect on single-and multi-threaded memory-intensive linear algebra software”. In: *Cluster Computing* 12.2 (2009), pp. 101–122.

[142] R. McDougall and J. Anderson. “Virtualization Performance: Perspectives and Challenges Ahead”. In: *SIGOPS Oper. Syst. Rev.* 44.4 (Dec. 2010), pp. 40–56.

[143] W. Huang, J. Liu, B. Abali, and D. K. Panda. “A case for high performance computing with virtual machines”. In: *Proceedings of the 20th annual international conference on Supercomputing*. ACM. 2006, pp. 125–134.

[144] K. Z. Ibrahim, S. Hofmeyr, and C. Iancu. “Characterizing the performance of parallel applications on multi-socket virtual machines”. In: *Proceedings of the 2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*. IEEE Computer Society. 2011, pp. 1–12.

[145] M. Grund, J. Schaffner, J. Krueger, J. Brunnert, and A. Zeier. “The Effects of Virtualization on Main Memory Systems”. In: *DaMoN 2010*. Proceedings of the Sixth International Workshop on Data Management on New Hardware. 2010, pp. 41–46.

[146] B. A. Kitchenham, S. L. Pfleeger, L. M. Pickard, P. W. Jones, D. C. Hoaglin, K. El Emam, and J. Rosenberg. “Preliminary guidelines for empirical research in software engineering”. In: *IEEE Transactions on software engineering* 28.8 (2002), pp. 721–734.

[147] R. Malhotra. *Empirical research in software engineering: concepts, analysis, and applications*. CRC press, 2016.

[148] A. Höfer and W. F. Tichy. "Status of empirical research in software engineering". In: *Empirical Software Engineering Issues. Critical Assessment and Future Directions*. Springer, 2007, pp. 10–19.

[149] M. Razavian, B. Paech, and A. Tang. "Empirical research for software architecture decision making: An analysis". In: *Journal of Systems and Software* 149 (2019), pp. 360–381.

[150] D. Etiemble. "45-year CPU evolution: one law and two equations". In: *arXiv preprint arXiv:1803.00254* (2018).

[151] C.-Q. Yang and B. P. Miller. "Critical path analysis for the execution of parallel and distributed programs". In: *ICDCS'88*. 1988, pp. 366–367.

[152] U. Drepper. *Memory part 7: Memory performance tools*. Nov. 2007. URL: https://lwn.net/Articles/257209/ (visited on 07/29/2020).

[153] G. Voron, G. Thomas, V. Quema, and P. Sens. "An interface to implement NUMA policies in the Xen hypervisor". In: *EuroSys'17*. 2017, pp. 453–467.

[154] B. Bui, D. Mvondo, B. Teabe, K. Jiokeng, L. Wapet, A. Tchana, G. Thomas, D. Hagimont, G. Muller, and N. Depalma. "When extended para-virtualization (XPV) meets NUMA". In: *EuroSys'19*. 2019, pp. 1–15.

[155] The Linux Kernel Archives. *NO_HZ: Reducing Scheduling-Clock Ticks*. URL: https://www.kernel.org/doc/Documentation/timers/%7BNO%5C_HZ%7D.txt (visited on 07/07/2021).

[156] S. Siddha, V. Pallipadi, and A. Ven. "Getting maximum mileage out of tickless". In: *Proceedings of the Linux Symposium*. Vol. 2. Citeseer. 2007, pp. 201–207.

[157] W. Xiong, S. Park, J. Zhang, Y. Zhou, and Z. Ma. "Ad Hoc Synchronization Considered Harmful". In: *OSDI'10*. Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation. Vancouver, BC, Canada: USENIX Association, 2010, pp. 163–176. URL: http://dl.acm.org/citation.cfm?id=1924943.1924955.

[158] W. Li. *KVM: X86: Add Paravirt TLB Shootdown*. Nov. 2017. URL: https://lwn.net/Articles/740363/ (visited on 07/07/2021).

[159] P. Monne Roger. *[v2,3/3] x86/tlb: use Xen L0 assisted TLB flush when available*. Jan. 2020. URL: https://patchwork.kernel.org/patch/11327803/ (visited on 07/29/2020).

[160] O. Sukwong and H. S. Kim. "Is co-scheduling too expensive for SMP VMs?" In: *Proceedings of the sixth conference on Computer systems*. ACM, 2011, pp. 257–272.

[161] X. Ding, P. Gibbons, and M. Kozuch. “A Hidden Cost of Virtualization When Scaling Multicore Applications”. In: *HotCloud 2013*. USENIX.

[162] J. Ahn, C. H. Park, and J. Huh. “Micro-sliced virtual processors to hide the effect of discontinuous cpu availability for consolidated systems”. In: *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*. IEEE. 2014, pp. 394–405.

[163] C.-C. Tu, M. Ferdman, C.-t. Lee, and T.-c. Chiueh. “A comprehensive implementation and evaluation of direct interrupt delivery”. In: *Acm Sigplan Notices* 50.7 (2015), pp. 1–15.

[164] L. Cheng, J. Rao, and F. C. M. Lau. “vScale: Automatic and Efficient Processor Scaling for SMP Virtual Machines”. In: *EuroSys ’16*. London, United Kingdom: ACM, 2016, 2:1–2:14. ISBN: 978-1-4503-4240-7. DOI: 10.1145/2901318.2901321. URL: http://doi.acm.org/10.1145/2901318.2901321.

[165] T. Merrifield and H. R. Taheri. “Performance Implications of Extended Page Tables on Virtualized x86 Processors”. In: VEE’16. 2016, pp. 25–35.

[166] S. Kashyap, C. Min, and T. Kim. “Scaling Guest OS Critical Sections with eCS”. In: Boston, MA: USENIX Association, 2018, pp. 159–172. ISBN: 978-1-931971-44-7. URL: https://www.usenix.org/conference/atc18/presentation/kashyap.

[167] J. T. Lim and J. Nieh. “Optimizing Nested Virtualization Performance Using Direct Virtual Hardware”. In: *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS ’20. Lausanne, Switzerland: Association for Computing Machinery, 2020, pp. 557–574. ISBN: 9781450371025. DOI: 10.1145/3373376.3378467. URL: https://doi.org/10.1145/3373376.3378467.

[168] N. Amit, A. Tai, and M. Wei. “Don’t Shoot down TLB Shootdowns!” In: *Proceedings of the Fifteenth European Conference on Computer Systems*. EuroSys ’20. Heraklion, Greece: Association for Computing Machinery, 2020. ISBN: 9781450368827. DOI: 10.1145/3342195.3387518. URL: https://doi-org.kuleuven.e-bronnen.be/10.1145/3342195.3387518.

[169] W. Jia, J. Shan, T. O. Li, X. Shang, H. Cui, and X. Ding. “vSMT-IO: Improving I/O Performance and Efficiency on {SMT} Processors in Virtualized Clouds”. In: *2020 {USENIX} Annual Technical Conference ({USENIX}{ATC} 20)*. 2020, pp. 449–463.

[170] S. K. Sadasivam, B. W. Thompto, R. Kalla, and W. J. Starke. “IBM Power9 processor architecture”. In: *IEEE Micro* 37.2 (2017), pp. 40–51.

[171] M. Danelutto, T. De Matteis, D. De Sensi, G. Mencagli, and M. Torquati. "P3ARSEC: towards parallel patterns benchmarking". In: *Proceedings of the Symposium on Applied Computing*. ACM. 2017, pp. 1582–1589.

[172] *InitializeCriticalSectionAndSpinCount function (synchapi.h)*. Oct. 2021. URL: https://docs.microsoft.com/en-us/windows/win32/api/synchapi/nf-synchapi-initializecriticalsectionandspincount (visited on 11/15/2021).

[173] *In-depth JVM-locking and concurrency*. URL: https://programmerall.com/article/4314276986/ (visited on 11/15/2021).

[174] *PAUSE*. URL: https://c9x.me/x86/html/file_module_x86_id_232.html (visited on 11/15/2021).

[175] M. Kerrisk. *futex(2)*. Aug. 2021. URL: https://man7.org/linux/man-pages/man2/futex.2.html (visited on 09/03/2021).

[176] S. Mittal. "A survey of techniques for architecting TLBs". In: *Concurrency and computation: practice and experience* 29.10 (2017).

[177] D. L. Black, R. F. Rashid, D. B. Golub, and C. R. Hill. "Translation lookaside buffer consistency: a software approach". In: *ACM SIGARCH Computer Architecture News* 17.2 (1989), pp. 113–122.

[178] S. Ghemawat and P. Menage. *Tcmalloc: Thread-caching malloc*. 2009.

[179] Emeryberger. *emeryberger/Malloc-Implementations*. July 2012. URL: https://github.com/emeryberger/Malloc-Implementations/tree/master/allocators/ptmalloc/ptmalloc2 (visited on 07/12/2021).

[180] S. Peter, A. Baumann, T. Roscoe, P. Barham, and R. Isaacs. "30 seconds is not enough! A study of operating system timer usage". In: *ACM SIGOPS Operating Systems Review* 42.4 (2008), pp. 205–218.

[181] Y. Etsion, D. Tsafrir, and D. G. Feitelson. "Effects of clock resolution on the scheduling of interactive and soft real-time processes". In: *Proceedings of the 2003 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*. 2003, pp. 172–183.

[182] J. Corbet. *Reinventing the timer wheel*. June 2015. URL: https://lwn.net/Articles/646950 (visited on 07/12/2021).

[183] R. Russell. *Unreliable Guide To Hacking The Linux Kernel*. 2005. URL: https://www.kernel.org/doc/htmldocs/kernel-hacking/index.html (visited on 07/12/2021).

[184] A. Golchin. "Control based tickless scheduling". PhD thesis. 2017.

[185] *Timer Interrupt Sources*. Mar. 2019. URL: https://wiki.osdev.org/Timer_Interrupt_Sources (visited on 07/12/2021).

[186] *[V4,4/4] Utilize the vmx preemption timer for tsc deadline timer*. June 2016. URL: https://patchwork.kernel.org/project/kvm/patch/1465852801-6684-5-git-send-email-yunhong.jiang@linux.intel.com/ (visited on 04/14/2021).

[187] M. C. Chehab and J. Lawall. "NO HZ: Reducing scheduling-clock ticks". In: *Linux Kernel Source Tree* (July 2020). URL: https://github.com/torvalds/linux/blob/master/Documentation/timers/no_hz.rst.

[188] H. Tang, Q. Koziol, S. Byna, J. Mainzer, and T. Li. "Enabling Transparent Asynchronous I/O using Background Threads". In: *2019 IEEE/ACM Fourth International Parallel Data Systems Workshop (PDSW)*. IEEE. 2019, pp. 11–19.

[189] L. Barroso, M. Marty, D. Patterson, and P. Ranganathan. "Attack of the killer microseconds". In: *Communications of the ACM* 60.4 (2017), pp. 48–54.

[190] E. VMware. *Timekeeping in VMware Virtual Machines*. 2008.

[191] T. Broomhead, L. Cremean, J. Ridoux, and D. Veitch. "Virtualize Everything but Time." In: *OSDI*. Vol. 10. 2010, pp. 1–6.

[192] S. D'Souza and R. Rajkumar. "QuartzV: Bringing Quality of Time to Virtual Machines". In: *2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE. 2018, pp. 49–61.

[193] A. Goel, L. Abeni, C. Krasic, J. Snow, and J. Walpole. "Supporting time-sensitive applications on a commodity OS". In: *ACM SIGOPS Operating Systems Review* 36.SI (2002), pp. 165–180.

[194] M. Aron and P. Druschel. "Soft timers: Efficient microsecond software timer support for network processing". In: *ACM Transactions on Computer Systems (TOCS)* 18.3 (2000), pp. 197–228.

[195] Y. Etsion, D. Tsafrir, and D. G. Feitelson. "Effects of clock resolution on the scheduling of interactive and soft real-time processes". In: *Proceedings of the 2003 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*. 2003, pp. 172–183.

[196] N. Amit. "Optimizing the TLB Shootdown Algorithm with Page Access Tracking". In: *2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 17)*. 2017, pp. 27–39.

[197] E. Rigtorp. *Latency implications of virtual memory*. July 2020. URL: https://rigtorp.se/virtual-memory/ (visited on 01/05/2022).

[198] Y. Kwon, H. Yu, S. Peter, C. J. Rossbach, and E. Witchel. "Coordinated and Efficient Huge Page Management with Ingens". In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. Savannah, GA: USENIX Association, Nov. 2016, pp. 705–721. ISBN: 978-1-931971-33-1. URL: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/kwon.

[199] *madvise(2)*. Mar. 2021. URL: http://man7.org/linux/man-pages/man2/madvise.2.html (visited on 07/12/2021).

[200] J. Evans. "A scalable concurrent malloc (3) implementation for FreeBSD". In: *Proc. of the bsdcan conference, ottawa, canada*. 2006.

[201] D. Rentas. *Evaluate the Fragmentation Effect of Different Heap Allocation Algorithms in Linux*. 2015.

[202] A. Wiggins and J. Langston. "Enhancing the scalability of memcached". In: *Intel document, unpublished, http://software. intel. com/en-us/articles/enhancing-the-scalability-of-memcached* (2012).

[203] Oracle. *Understanding Memory Management*. Jan. 2010. URL: https://docs.oracle.com/cd/E13150_01/jrockit_jvm/jrockit/geninfo/diagnos/garbage_collect.html (visited on 07/12/2021).

[204] S. Sangappa, K. Palaniappan, and R. Tollerton. "Benchmarking Java against C/C++ for interactive scientific visualization". In: *Proceedings of the 2002 joint ACM-ISCOPE conference on Java Grande*. 2002, pp. 236–236.

[205] L. Prechelt. "An empirical comparison of seven programming languages". In: *Computer* 33.10 (2000), pp. 23–29.

[206] P. Kulkarni, H. Kailash, V. Shankar, S. Nagarajan, and D. Goutham. "Programming languages: A comparative study". In: *Information Security Research Lab, NITK, Surathkal* (2008).

[207] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio. "An updated performance comparison of virtual machines and linux containers". In: *2015 IEEE international symposium on performance analysis of systems and software (ISPASS)*. IEEE. 2015, pp. 171–172.

[208] *mallopt(3)*. Mar. 2021. URL: https://man7.org/linux/man-pages/man3/mallopt.3.html (visited on 01/14/2022).

[209] R. Liu and H. Chen. "SSMalloc: a low-latency, locality-conscious memory allocator with stable performance scalability". In: *Proceedings of the Asia-Pacific Workshop on Systems*. 2012, pp. 1–6.

[210] B. F. Romanescu, A. R. Lebeck, D. J. Sorin, and A. Bracy. "UNified instruction/translation/data (UNITD) coherence: One protocol to rule them all". In: *HPCA 2010 The Sixteenth International Symposium on High-Performance Computer Architecture*. IEEE. 2010, pp. 1–12.

[211] A. Bhattacharjee, D. Lustig, and M. Martonosi. "Shared last-level TLBs for chip multiprocessors". In: *2011 IEEE 17th International Symposium on High Performance Computer Architecture*. IEEE. 2011, pp. 62–63.

[212] M. K. Kumar, S. Maass, S. Kashyap, J. Veselỳ, Z. Yan, T. Kim, A. Bhattacharjee, and T. Krishna. "Latr: Lazy Translation Coherence". In: *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM. 2018, pp. 651–664.

[213] E. Zurich. *The Barrelfish Operating System*. Oct. 2018. URL: `http://www.barrelfish.org/index.html` (visited on 01/21/2022).

[214] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, M. F. Kaashoek, R. T. Morris, A. Pesterev, L. Stein, M. Wu, Y.-h. Dai, et al. "Corey: An Operating System for Many Cores." In: *OSDI 2008*. Vol. 8. 2008, pp. 43–57.

[215] A. T. Clements, M. F. Kaashoek, and N. Zeldovich. "RadixVM: Scalable address spaces for multithreaded applications". In: *Proceedings of the 8th ACM European Conference on Computer Systems*. 2013, pp. 211–224.

[216] S. Schneider, C. D. Antonopoulos, and D. S. Nikolopoulos. "Scalable locality-conscious multithreaded memory allocation". In: *Proceedings of the 5th international symposium on Memory management*. 2006, pp. 84–94.

[217] *What is data deduplication*. 2022. URL: `https://www.netapp.com/data-management/what-is-data-deduplication/#:~:text=Data%20deduplication%20is%20a%20process,data%20is%20written%20to%20disk.` (visited on 02/02/2022).

[218] N. Koksharov. *What is a ring buffer?* 2021. URL: `https://redisson.org/glossary/ring-buffer.html` (visited on 02/01/2022).

[219] R. Johnson, M. Athanassoulis, R. Stoica, and A. Ailamaki. "A new look at the roles of spinning and blocking". In: *Proceedings of the Fifth International Workshop on Data Management on New Hardware*. 2009, pp. 21–26.

[220] *Using mutexes*. 2020. URL: `https://www.ibm.com/docs/en/aix/7.2?topic=programming-using-mutexes` (visited on 02/03/2022).

[221] *sem_overview(7)*. June 2020. URL: `https://man7.org/linux/man-pages/man7/sem%5C_overview.7.html` (visited on 02/03/2022).

[222] *Using Condition Variables*. 2010. URL: https://docs.oracle.com/cd/E19455-01/806-5257/6je9h032r/index.html (visited on 02/03/2022).

[223] *Monitors and Condition Variables*. URL: https://cseweb.ucsd.edu/classes/sp17/cse120-a/applications/ln/lecture8.html (visited on 02/03/2022).

[224] *Synchronized Methods*. 2021. URL: https://docs.oracle.com/javase/tutorial/essential/concurrency/syncmeth.html (visited on 02/03/2022).

[225] A. Kumar. *Monitor And Lock In C#*. May 2019. URL: https://www.c-sharpcorner.com/UploadFile/de41d6/monitor-and-lock-in-C-Sharp/ (visited on 02/03/2022).

[226] J. Hughes. "Why functional programming matters". In: *The computer journal* 32.2 (1989), pp. 98–107.

[227] E. Bainomugisha, A. L. Carreton, T. v. Cutsem, S. Mostinckx, and W. d. Meuter. "A Survey on Reactive Programming". In: *ACM Comput. Surv.* 45.4 (Aug. 2013). ISSN: 0360-0300. DOI: 10.1145/2501654.2501666. URL: https://doi.org/10.1145/2501654.2501666.

[228] *Parallelism*. 2021. URL: https://docs.oracle.com/javase/tutorial/collections/streams/parallelism.html (visited on 02/03/2022).

[229] *Run MATLAB on multicore and multiprocessor machines*. 2022. URL: https://www.mathworks.com/discovery/matlab-multicore.html (visited on 02/13/2022).

[230] E. Rigtorp. *Correctly implementing a spinlock in C++*. Apr. 2020. URL: https://rigtorp.se/spinlock/ (visited on 02/04/2022).

[231] D. Loshin. *Business intelligence: the savvy manager's guide*. Newnes, 2012.

[232] E. Paraschiv. *Introduction to Thread Pools in Java*. Jan. 2022. URL: https://www.baeldung.com/thread-pool-java-and-guava (visited on 02/08/2022).

[233] S. Schildermans and K. Aerts. "Wolfram for data processing and visualization". In: *Draft Proceedings of the 29th Symposium on Implementation and Application of Functional Languages (IFL 2017)*. Nicolas Wu, University of Bristol; Bristol. 2017.

Biography

I obtained my Master's degree in Engineering Technology, Electronics-ICT summa cum laude with congratulations from the examination committee from KU Leuven and UHasselt in 2017. Ever since, up until the time of writing this dissertation, I have been working on the Ph. D. project documented here.

My main research interests should be evident from the work before you. However, since I am highly motivated and curious by nature, I have accrued sizeable knowledge in other fields related to computer science over the years, including functional programming, software architectures, embedded systems, etc. Given the opportunity, I would gladly continue down this path of becoming a true Swiss army knife of software engineering: versatile, effective and indispensable in any good project manager's toolkit.

Besides software development, I have a passion for music and martial arts. I play electric guitar and have been practicing judo for the majority of my life. A few years ago I picked up kickboxing as well. Those formal hobbies aside, my favorite passtime will likely always remain devising novel and creative methods to annoy my girlfriend followed by empirically studying how she reacts to them.

List of publications

- S. Schildermans and K. Aerts. "Wolfram for data processing and visualization". In: *Draft Proceedings of the 29th Symposium on Implementation and Application of Functional Languages (IFL 2017)*. Nicolas Wu, University of Bristol; Bristol. 2017;
- S. Schildermans and K. Aerts. "Towards High-Level Software Approaches to Reduce Virtualization Overhead for Parallel Applications". In: *2018 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE. 2018, pp. 193–197;
- S. Schildermans et al. "Ptlbmalloc2: Reducing TLB Shootdowns with High Memory Efficiency". In: *ISPA-BDCloud-SocialCom-SustainCom 2020* (2020), pp. 76–83;
- S. Schildermans et al. "Virtualization Overhead of Multithreading in X86 State-of-the-Art & Remaining Challenges". In: *IEEE Transactions on Parallel and Distributed Systems* 32.10 (2021), pp. 2557–2570;
- S. Schildermans et al. "Paratick: Reducing Timer Overhead in Virtual Machines". In: *50th International Conference on Parallel Processing*. 2021, pp. 1–10.

FACULTY OF ENGINEERING TECHNOLOGY
DEPARTMENT OF COMPUTER SCIENCE
ACRO-FUNTTOP
Wetenschapspark 27
3590 Diepenbeek
stijn.schildermans@kuleuven.be
https://iiw.kuleuven.be/onderzoek/acro