

# **Runtime Minimization of Scalable Network Emulation**

Von der Fakultät Informatik, Elektrotechnik und  
Informationstechnik der Universität Stuttgart  
zur Erlangung der Würde eines Doktors der  
Naturwissenschaften (Dr. rer. nat.) genehmigte Abhandlung

Vorgelegt von

**Andreas Grau**

aus Bendorf am Rhein

Hauptberichter: Prof. Dr. rer. nat. Dr. h. c. Kurt Rothermel

Mitberichter: Prof. Dr.-Ing. Klaus Wehrle

Mitprüfer: Prof. Dr. rer. nat. Otto Eggenberger

Tag der mündlichen Prüfung: 8. Oktober 2012

Institut für Parallele und Verteilte Systeme (IPVS)  
der Universität Stuttgart  
2012



# Acknowledgements

First of all, I want to thank my PhD advisor Kurt Rothermel for the opportunity to accomplish my research in his group. I would like to thank him for the support and the advice he provided me in prolific research discussions. I also want to thank Klaus Wehrle for his efforts as my co-advisor.

During my research at the distributed systems group I worked together with a lot of inspiring research colleagues. Their encouraging, constructive, and sometimes reprehensive but always fair-minded feedback helped me a lot to improve my research. Many thanks go to Klaus Herrmann, Steffen Maier, Harald Weinschrott, Lars Geiger, Faraz Memon, Björn Schilling, Marius Wernke, Stefan Föll, and of course all the other members of the research group.

A big thanks goes to all my students supporting me during the development of the NET system. Their effort and contribution to this project have made this thesis possible. Special thanks to Alexander Egorenkov, Frank Schuh, Frederik Pakai, Markus Schirmer, Kai Zhou, Sebastian Bartmann and of course the whole development team of "Mission Control".

Many thanks go to the German Science Foundation, that promoted my research under grant "DFG-GZ RO 1086/9-1" and provided me the chance to present my results to the research community on various international conferences.

Special thanks go to my brothers Michael and Alexander for their valuable comments to improve the quality of this work.

Last but not least, I want to thank my mother of her continuous patience and support which has enabled me to fully focus on the work for this thesis.



# List of Abbreviations

APIC	Advanced Programmable Interrupt Controller
CPU	Central Processing Unit
CSP	Constraint Satisfaction Problem
DES	Discrete Event Simulation
DSL	Digital Subscriber Line
FIFO	First In, First Out
GB	Gigabyte
Gbps	Gigabit per Second
GHz	Gigahertz
GUI	Graphical User Interface
host-os	Host Operating System
HPC	High Performance Computing
kB	Kilobyte
kbps	Kilobit per Second
I/O	Input/Output
ICMP	Internet Control Message Protocol
IEEE	Institute of Electrical and Electronics Engineers
IP	Internet protocol
LAN	Local Area Network
LKM	Linux Kernel Module
MAC	Media Access Control
MANET	Mobile Ad-Hoc Network
MB	Megabyte
Mbps	Megabit per Second
MCps	Million Cycles per Second
MTU	Maximum Transmission Unit
ms	Millisecond
$\mu$ s	Millisecond
NET	Network Emulation Testbed

---

NFS	Network File System
NIC	Network Interface Card
OLSR	Optimized Link State Routing
PC	Personal Computer
PDES	Parallel Discrete Event Simulation
QoS	Quality of Service
RAM	Random Access Memory
RDMA	Remote Direct Memory Access
RFC	Request for Comment
RTT	Round Trip Time
RF	Radio Frequency
SuT	Software under Test
TCP	Transmission Control Protocol
TDF	Time Dilation Factor
TOS	Type of Service
TSC	Time Stamp Counter
TTL	Time to Live
UDP	User Datagram Protocol
UML	User Mode Linux
veth	Virtual Ethernet Device Driver
VLAN	Virtual Local Area Network
VM	Virtual Machine
VMDq	Virtual Machine Device Queue
VMM	Virtual Machine Monitor
vnode	Virtual Node

# Contents

<b>Acknowledgments</b>	<b>3</b>
<b>List of Abbreviations</b>	<b>5</b>
<b>Abstract</b>	<b>11</b>
<b>Deutsche Zusammenfassung</b>	<b>13</b>
<b>1 Introduction</b>	<b>25</b>
1.1 Motivation . . . . .	25
1.2 Contributions . . . . .	28
1.3 Structure . . . . .	29
<b>2 Background</b>	<b>31</b>
2.1 Emulation of Link Characteristics . . . . .	31
2.2 Node Virtualization . . . . .	34
2.2.1 Virtual Machines . . . . .	34
2.2.2 Virtual Protocol Stacks . . . . .	36
2.3 Time Virtualization . . . . .	37
2.3.1 Implementation Approaches for Virtual Time . . . . .	39
2.3.2 Approaches to Virtual Time Representation . . . . .	40
2.4 Performance Evaluation Tools . . . . .	41
2.4.1 Real World Testbeds . . . . .	41
2.4.2 Network Simulation . . . . .	42
2.4.3 Emulation-based Testbeds . . . . .	43
<b>3 System Overview</b>	<b>45</b>
3.1 Network Emulation Testbed . . . . .	45
3.2 General Architecture . . . . .	47
3.3 Application Case Studies . . . . .	49

<b>4 Efficient Node and Time Virtualization</b>	<b>51</b>
4.1 Architecture for Efficient Virtual Time . . . . .	52
4.1.1 Application Model and Hardware Model . . . . .	52
4.1.2 Hybrid Virtualization Architecture . . . . .	54
4.1.3 Efficient Network Access . . . . .	57
4.2 Fine-Grained Timer for NETshaper . . . . .	60
4.3 Multiplexing of Virtual Links . . . . .	62
4.3.1 VLAN-based Network Emulation . . . . .	62
4.3.2 Related Work . . . . .	64
4.3.3 Concepts for Scalable Link Multiplexing . . . . .	65
4.4 Adaptive Virtual Time . . . . .	69
4.4.1 Related Work . . . . .	70
4.4.2 TDF Adaptation Process . . . . .	71
4.5 Evaluation . . . . .	78
4.5.1 Architecture for Efficient Virtual Time . . . . .	78
4.5.2 Adaptive Virtual Time . . . . .	83
4.6 Summary . . . . .	91
<b>5 Experiment Configuration</b>	<b>93</b>
5.1 Experiment Specification . . . . .	95
5.1.1 Textual Experiment Specification . . . . .	95
5.1.2 Graphical Experiment Specification . . . . .	98
5.2 Testbed Model . . . . .	99
5.3 Initial Node Placement . . . . .	103
5.3.1 Related Work . . . . .	103
5.3.2 Formal Problem Statement . . . . .	105
5.3.3 Placement Algorithms . . . . .	106
5.4 Dynamic Reconfiguration . . . . .	110
5.4.1 Related Work . . . . .	110
5.4.2 Migration Support for Network Emulation . . . . .	112
5.4.3 Migration Cost Model . . . . .	114
5.4.4 Reconfiguration Concepts . . . . .	115
5.5 Evaluation . . . . .	118
5.5.1 Testbed Model . . . . .	118
5.5.2 Initial Node Placement . . . . .	122
5.5.3 Dynamic Reconfiguration . . . . .	127
5.6 Summary . . . . .	134

<b>6 Summary</b>	<b>135</b>
6.1 Conclusions . . . . .	135
6.2 Promising Research Directions . . . . .	138
<b>List of Figures</b>	<b>139</b>
<b>List of Tables</b>	<b>141</b>
<b>List of Algorithms</b>	<b>142</b>
<b>Publications</b>	<b>143</b>
<b>Supervised Student Theses</b>	<b>145</b>
<b>Bibliography</b>	<b>146</b>



# Abstract

Network emulation constitutes an approved methodology to evaluate the performance of distributed applications and communication protocols. The approach of network emulation models computer networks by connecting instances of the *Software under Test* (SuT), representing routers and hosts, using a distributed emulation tool. The emulation tool allows for specifying the parameters of these connections, like bandwidth, delay and loss rate. Therefore, network emulation combines the benefits of network simulation, like controllability and repeatability of network experiments, with the benefits of real world testbeds, like accuracy and realism of running the unmodified implementations of the SuT.

Recently, researchers have spent much effort to increase the scalability of network emulation to allow for the evaluation of distributed systems in large network topologies with thousands of network nodes. Basically two concepts are introduced to reach that goal: node and time virtualization. Node virtualization allows for partitioning the physical nodes of an emulation testbed to run multiple instances of the SuT on each physical node. However, the available resources of the physical nodes limit the number of virtual nodes that can be executed by the physical nodes without overloading the hardware. This overload can be avoided by applying the concept of time virtualization. Executing the network experiment with a virtual time, that runs a factor slower than real time, allows for increasing the resources like CPU and network capacity of the testbed by the same factor. However, the runtime of network experiments is also increased by that factor.

The goal of this work is to reduce the runtime of network experiments and, thus, increase the satisfaction of testbed users and testbed operators. Therefore, this thesis makes the following contributions. First, we present an efficient emulation architecture for testbeds with multi-core processors that provides node and time virtualization and minimizes CPU and memory consumption as well as the communication overhead. Second, we introduce the new concept of adaptive virtual time, that allows for dynamically adjusting the speed of the experiment to the resource requirements during the

## Contents

---

experiment runtime. Using this approach, the experiment can run with an increased speed during periods of low resource requirements. Third, we provide an accurate testbed model to capture the resource requirements of an experiment. Fourth, based on this model, we introduce an approach to calculate an initial placement of virtual nodes onto the physical nodes that minimizes the experiment runtime based on the experiment specification. Finally, in order to react on varying resource requirements during the experiment, we provide an approach to adapt the placement during the running experiment based on transparent migration of virtual nodes to further reduce the experiment runtime.

The developed concepts are implemented and integrated in our Network Emulation Testbed (NET). Detailed evaluations of our prototype show the efficiency and effectiveness of our concepts to minimize the runtime of experiments based on network emulation.

# Deutsche Zusammenfassung

## Laufzeitminimierung von skalierbarer Rechnernetz-Emulation

### 1 Einleitung

Der Softwaretest ist ein integraler Bestandteil des Entwicklungsprozesses von Software [Pre92]. Hierbei wird neben den funktionalen Anforderungen auch die Leistungsfähigkeit der zu testenden Software überprüft. Solche Leistungstests werden auch in der Forschung bei der Evaluation neu entwickelter verteilter Software (verteilte Anwendungen und Kommunikationsprotokolle) eingesetzt. Die Leistungsfähigkeit der verteilten Software hängt hierbei entscheidend von der Umgebung ab, in der sie ausgeführt wird. Daher erfordert eine akkurate Leistungsprüfung die Berücksichtigung der Zielumgebung der zu testenden Software.

Zur Leistungsprüfung von verteilter Software existieren vier Methodiken [Liu08a]: analytische Modelle [Gro06], Realwelt-Testumgebung [CCR<sup>+</sup>03, ABKM01], Netzwerksimulation [Liu08b, Fuj89, Ril03] und Netzwerkemulation [AC06, HRS<sup>+</sup>08, GMHR08]. Während bei der Leistungsprüfung mittels analytischer Modelle und der Netzwerksimulation Modelle der zu testenden Software evaluiert werden, kann bei Realwelt-Testumgebungen und bei der Netzwerkemulation unmodifizierte Software auf ihre Leistungsfähigkeit überprüft werden. Im Gegensatz zu Realwelt-Testumgebungen ist bei der Evaluation auf Basis der Netzwerkemulation die Zielumgebung der Software frei wählbar und nicht auf die Topologie der Testumgebung beschränkt.

Bei der Methode der Netzwerkemulation wird die Zielumgebung als Netzwerk von Knoten (Router und Endsysteme) modelliert. Auf diesen Knoten werden die Instanzen der zu testenden Software ausgeführt. Mittels konfigurierbarer Netzwerkhardware und Softwarewerkzeugen [Riz97, HR02] kann die Topologie des Netzwerks sowie dessen Eigenschaften (z.B. Bandbreite, Verzögerung und Verlustrate) bereitgestellt werden.

Diese Netzwerkexperimente werden auf Clustern aus handelsüblicher PC-Hardware ausgeführt. Das Konzept der *Knotenvirtualisierung* [WSG02, VYW<sup>+</sup>02, JX03, Mai11] erlaubt es, mehrere Instanzen der zu testenden Software (virtueller Knoten) auf jedem der PC-Knoten des Clusters (physischer Knoten) auszuführen und somit die Skalierbarkeit der Netzwerkemulation zu steigern.

Die Anzahl der virtuellen Knoten pro physischem Knoten ist durch dessen Leistung limitiert. Eine zu hohe Anzahl an virtuellen Knoten pro physischem Knoten führt zu dessen Überlastung, was schließlich zu einer Verfälschung der Experimentergebnisse führen kann. Das Konzept der *virtuellen Zeit* [CFH<sup>+</sup>80, GYM<sup>+</sup>06] erlaubt es, ein Experiment langsamer als Echtzeit auszuführen und somit die Last der physischen Knoten in der Emulationsumgebung zu reduzieren. Beispielsweise kann durch Verlangsamung des Experiments um den Faktor 10 ein reales 10 Mbps Netzwerk genutzt werden, um 100 Mbps in der Emulation zu übertragen. Der Quotient der Experimentzeit (virtuelle Zeit) und der Realzeit wird Zeitdehnungsfaktor *TDF* (engl. *time dilation factor*) genannt. Somit kann ein an dieses reale 10 Mbps Netzwerk angeschlossener physischer Knoten zehn virtuelle Knoten ausführen, wobei jeder dieser virtuellen Knoten 10 Mbps nutzen kann. Existierende Systeme [GYM<sup>+</sup>06, ELL09] mit konstantem TDF wählen den TDF sehr konservativ, um Überlast auch in Zeiten von Spitzenlast zu vermeiden. Daher werden bei Experimenten mit wechselnden Ressourcenanforderungen während einem Großteil der Experimentlaufzeit die physischen Ressourcen nur teilweise ausgelastet. Die Konsequenz ist eine suboptimale Laufzeit des Experiments, da die Ausführungsgeschwindigkeit der Experimente in diesen Teilen ohne Überlastung der physischen Ressourcen beschleunigt werden könnte. Zur Vermeidung von Überlast und gleichzeitiger Optimierung der Experimentlaufzeit wird in dieser Arbeit das Konzept der *adaptiven virtuellen Zeit* [GMHR08] eingeführt. Die Grundidee ist hierbei die Überwachung der Last der physischen Knoten und die dynamische Anpassung der Ausführungsgeschwindigkeit des Experiments, basierend auf der aktuellen Lastsituation [GHR09a].

Bei der Nutzung von adaptiver virtueller Zeit wird die Ausführungsgeschwindigkeit und somit die Laufzeit eines Experiments durch den am stärksten belasteten physischen Knoten bestimmt. Diese Last hängt stark von der Platzierung der virtuellen Knoten auf die physischen Knoten ab. Existierende Verfahren [RAL03, ZN03, LC04, CBMP04, LLXC05] zur Berechnung einer solchen Platzierung berücksichtigen keine Zeitvirtualisierung. Sie gehen von physischen Knoten mit festgelegten Kapazitäten aus und berechnen eine Platzierung, welche diese Kapazitäten nicht übersteigt. Bei der Wahl einer geeigneten Ausführungsgeschwindigkeit kann ein Experiment mit beliebiger Platzierung ohne Überlastung der Ressourcen ausgeführt werden. Allerdings führt eine Verlangsamung

der Ausführungsgeschwindigkeit zu einer Erhöhung der Experimentlaufzeit. Daher wird ein Platzierungsalgorithmus benötigt, der eine Platzierung berechnet, welche die Last des am meisten belasteten Knoten minimiert. Im Folgenden wird die Platzierung der virtuellen Knoten zu Beginn des Experiments als *initiale Platzierung* bezeichnet.

Selbst unter der Annahme einer optimalen initialen Platzierung können wechselnde Ressourcenanforderungen von virtuellen Knoten während der Experimentlaufzeit zu temporär suboptimalen Platzierungen und somit zu einer suboptimalen Ausführungs geschwindigkeit des Experiments führen. Durch Überwachen der Last der virtuellen und physischen Knoten können diese Situationen erkannt und eine optimierte Platzierung berechnet werden. Die Migration von virtuellen Knoten erlaubt es, eine optimierte Platzierung in der Emulationsumgebung herzustellen. Hierdurch wird eine Balancierung der Last zwischen den physischen Knoten erreicht und somit Spitzenlast auf einzelnen physischen Knoten vermieden. In Kombination mit dem Konzept der adaptiven virtuellen Zeit erlaubt es die Migration von virtuellen Knoten, die Laufzeit von Netzwerkexperimenten mit wechselnden Ressourcenanforderungen zu reduzieren.

Diese Arbeit leistet folgende sechs Hauptbeiträge: eine leichtgewichtige Architektur zur Knoten- und Zeitvirtualisierung mit Unterstützung von Multikern-Architekturen, eine auf Epochen basierende adaptive Zeitvirtualisierung zur Minimierung der Experiment laufzeit durch Maximierung der Ressourcenauslastung bei gleichzeitigem minimalen Synchronisierungsaufwand, ein generisches Kostenmodell für die Kommunikation zwischen virtuellen Knoten zur Abschätzung der Laufzeit von Netzwerkexperimen ten, ein Algorithmus zur Berechnung einer initialen Platzierung zur Minimierung der Experimentlaufzeit, eine Architektur zur effizienten Unterstützung von Migrationen virtueller Knoten zur Laufzeit von Experimenten, ein Verfahren zur Minimierung der Experimentlaufzeit durch Umplatzierung von virtuellen Knoten.

Die Arbeit ist folgendermaßen gegliedert: in Kapitel 2 werden die Grundlagen dieser Arbeit betrachtet und die Grundkomponenten skalierbarer Emulationsumgebungen vorgestellt. Kapitel 3 gibt einen Überblick über das zugrunde liegende Emulationssystem *.NET* sowie dem Zusammenspiel der entwickelten Konzepte. In Kapitel 4 wird eine Architektur zur effizienten Knoten- und Zeitvirtualisierung vorgestellt und das Konzept der adaptiven virtuellen Zeit zur Minimierung der Experimentlaufzeit bei gleichzeitiger Verhinderung von Überlast eingeführt. Kapitel 5 befasst sich mit der Minimierung der Experimentlaufzeit durch eine laufzeitoptimale Platzierung von virtuellen Knoten. Hierbei werden Mechanismen zur initialen Knotenplatzierung zu Beginn des Experiments und der dynamischen Knotenplatzierung während eines Experimentlaufs betrachtet.

Die Arbeit schließt mit einer Zusammenfassung und der Diskussion möglicher zukünftiger Arbeiten in Kapitel 6.

## 2 Grundlagen

In Kapitel 2 werden zunächst die Grundkomponenten skalierbarer Emulationsumgebungen betrachtet. Diese Komponenten umfassen Werkzeuge zur Emulation von Verbindungen zwischen virtuellen Knoten, Konzepte zur Knotenvirtualisierung sowie Konzepte zur Virtualisierung der Zeit. Im Anschluss werden aus diesen Grundkomponenten bestehende Werkzeuge zur Leistungsbewertung verteilter Systeme vorgestellt.

Zur Emulation von Verbindungen zwischen virtuellen Knoten werden unterschiedliche Methoden verwendet. In Realwelt-Testumgebungen [CCR<sup>+</sup>03, BL03] werden die Eigenschaften der Verbindungen durch Eigenschaften des zugrunde liegenden Netzwerks bestimmt. Die Emulation von beliebigen Netzeigenschaften kann durch verteilte Emulationswerkzeuge [HR02, Riz97] oder durch den Einsatz eines angeschlossenen Netzwerksimulators [Fal99, SU03, TRR08, WSHW08, LLH09] erreicht werden. Je nach Position des Emulationswerks beziehungsweise dem Ort der Anbindung des Netzwerksimulators im Protokollstapel, wird hierbei zu testende Software auf der Anwendungs- und Transportschicht [Riz97] oder zusätzlich auf Netzwerkschicht [HR02, CS03, Hem05, GMHR08, CR10] ermöglicht.

Zur Knotenvirtualisierung kommen in Emulationssystemen zwei verschiedene Virtualisierungstechniken zum Einsatz: virtuelle Maschinen [SM79, Cre81] und virtuelle Protokollstapel [KHS<sup>+</sup>03]. Virtuelle Maschinen erlauben es, jedem virtuellen Knoten sein eigenes Betriebssystem inklusive aller Komponenten, wie dem Protokollstapel, auszuführen und sind daher sehr flexibel. Im Unterschied zu virtuellen Maschinen teilen sich virtuelle Knoten, im Falle virtueller Protokollstapel, ein gemeinsames Betriebssystem. Hierbei nutzt jeder virtuelle Knoten eine eigene Instanz des Protokollstapels. Durch Techniken wie Partitionierung der Namensräume sowie der Virtualisierung des Dateisystems [BDM99, Sch00, KW00] können virtuelle Knoten von einander isoliert werden. Der Vergleich des notwendigen Ressourcenbedarfs beider Virtualisierungstechniken bezüglich Speicherbedarf und Kommunikationskosten zeigt, dass virtuelle Maschinen einen um eine Größenordnung höheren Aufwand als virtuelle Protokollstapel besitzen [BQ06a, MGWR07, CGMV07, SPF<sup>+</sup>07].

Das Konzept der virtuellen Zeit erlaubt es, Netzwerkexperimente schneller oder langsamer als Echtzeit auszuführen. Während eine schnellere Ausführung beispielsweise

genutzt werden kann, um das zukünftige Verhalten von Schadsoftware zu untersuchen [CWdO<sup>+</sup>06], erlaubt es eine langsamere Ausführung [CFH<sup>+</sup>80, GYM<sup>+</sup>06], die Ressourcen der Testumgebung künstlich zu vergrößern und somit eine Ressourcenüberlastung zu verhindern. Virtuelle Zeit kann auf drei Arten bereitgestellt werden: durch Erweiterung der Anwendungsschnittstelle des Betriebssystems [ZN11, BF11], durch Einführung einer virtuellen Zeit im Betriebssystems [WK02] und durch Ausführung der zu testenden Software innerhalb einer um virtuelle Zeit erweiterten virtuellen Maschine [GYM<sup>+</sup>06]. Letzteres stellt virtuelle Zeit der zu testenden Software transparent dar und erfordert keine Anpassungen des Betriebssystems.

Werkzeuge zur Leistungsbewertung verteilter Anwendung werden typischerweise in Realwelt-Testumgebungen, Netzwerkemulatoren und Netzwerksimulatoren unterschieden. Traditionell führen Realwelt-Testumgebungen und Netzwerkemulatoren die Experimente in Realzeit durch, während Netzwerksimulatoren eine virtuelle Zeit nutzen. Im Gegensatz zu Realwelt-Testumgebungen erlauben Netzwerkemulatoren eine Leistungsbewertung von Software in beliebigen Netzwerktopologien. Es zeigt sich allerdings, dass diese Grenzen durch neuere Entwicklungen unscharf werden. Wie bereits erwähnt wurde, nutzen aktuelle Netzwerkemulatoren virtuelle Zeit um Überlast zu verhindern [GVV08, GHR12]. Gleichzeitig werden Realzeit-Scheduler in Netzwerksimulatoren verwendet, um realen Netzwerkverkehr in der Simulation zu verarbeiten [Fal99, SU03, TRR08, LLH09, KP09, AOC<sup>+</sup>10].

### 3 Systemüberblick

In Kapitel 3 wird zunächst die für diese Arbeit als Basis dienende Emulationsumgebung *NET* (*Network Emulation Testbed*) vorgestellt. Darauf folgend wird die entwickelte Architektur zur Minimierung der Laufzeit von Netzwerkexperimenten eingeführt. Als Abschluss des Kapitels werden Anwendungsbeispiele des im Rahmen dieser Arbeit entstandenen Prototyps diskutiert.

Die Konzepte und Methoden dieser Arbeit sind Teil des NET-Projekts der Abteilung Verteilte Systeme der Universität Stuttgart. Hierbei dienen die Forschungsergebnisse früherer Arbeiten in diesem Projekt als Basis. Die Arbeit von Herrscher [Her05] befasst sich mit der Entwicklung des Emulationswerkzeugs *NETshaper* [HR02] zur akkurate Emulation von drahtlosen und drahtgebundenen Netzwerken. Die Arbeit von Maier [Mai11] erweitert die Emulationsumgebung NET um das Konzept der virtuellen Knoten auf Basis von virtuellen Protokollstapeln.

Die Komponenten dieser Arbeit sind auf mehrere Knoten der Emulationsumgebung verteilt. Zunächst wird vom Benutzer mittels eines Spezifikationswerkzeugs [Gra11, VGR<sup>+</sup>11] eine Experimentbeschreibung erstellt. Diese Beschreibung umfasst die zu testende Software, die zu emulierende Netzwerktopologie sowie eine Beschreibung der zu verwendenden Experimentumgebung. Basierend auf diesen Beschreibungen berechnet ein Knoten in der Rolle des Koordinators mittels des entwickelten Platzierungsalgorithmus *NETplace* eine initiale Knotenplatzierung. Gemäß dieser Platzierung werden die virtuellen Knoten auf die physischen Knoten verteilt, die virtuelle Netzwerktopologie aufgebaut und die zu testende Software auf den virtuellen Knoten gestartet. Zur Verhinderung von Ressourcenengpässen wird die Last der physischen Knoten überwacht und an den Koordinator gemeldet. Auf Basis dieser Lastinformationen passt der Koordinator die Ausführungsgeschwindigkeit des Experiments an. Um die Platzierung der Knoten an den sich ändernden Ressourcenbedarf der virtuellen Knoten anzupassen, wird die Last der virtuellen Knoten ebenfalls überwacht und an den Koordinator gesendet. Auf Basis dieser Lastinformationen berechnet der Koordinator (*NETbalance*) eine optimierte Platzierung. Bei einer erwarteten Reduzierung der Experimentlaufzeit durch die optimierte Platzierung wird diese in der Testumgebung umgesetzt.

## 4 Effiziente Knoten- und Zeitvirtualisierung

Kapitel 4 beschäftigt sich mit Konzepten zur Erhöhung der Skalierbarkeit der Netzwerkemulation bezüglich der Größe der Szenarien und dem Ressourcenverbrauch. Hierzu wird erstens eine Architektur zur effizienten Knoten- und Zeitvirtualisierung auf Multikern-Architekturen vorgestellt, welche den Speicherbedarf sowie den Kommunikationsaufwand minimiert. Zweitens wird das eingesetzte Emulationswerkzeug *NETshaper* für den Einsatz in einer zeitvirtualisierten Emulationsumgebung angepasst. Drittens wird ein skalierbarer Multiplexing-Ansatz zur Emulation einer Vielzahl an virtuellen Verbindungen mittels einer einzigen physischen Netzwerkkarte vorgestellt. Schließlich wird das Konzept der adaptiven virtuellen Zeit eingeführt. Das Kapitel schließt mit einer Leistungsbewertung der vorgestellten Konzepte. Teile der vorgestellten Konzepte wurden bereits zuvor in Tagungsbänden [GMHR08, GHR09a, GHR10] sowie in einem Fachblatt [GHR12] veröffentlicht.

Virtuelle Maschinen können der zu testenden Software eine virtuelle Zeit transparent bereitstellen. Um den Mehraufwand für die Knotenvirtualisierung bei gleichzeitiger Zeitvirtualisierung zu minimieren, werden die Ressourcen der virtuellen Maschine

durch virtuelle Protokollstapel partitioniert (virtuelle Knoten). Hierdurch können mittels Referenzweitergabe die Kommunikationskosten und mittels geteilten Zugriffs auf Systembibliotheken der Speicherbedarf reduziert werden [GMHR08]. Zur Unterstützung von Multikern-Architekturen wird auf jedem Kern der CPU eine virtuelle Maschine ausgeführt. Die Zuweisung einer CPU pro virtueller Maschine ermöglicht es, einerseits auf teure Synchronisation mehrerer CPUs innerhalb einer virtuellen Maschine zu verzichten und andererseits die Last der virtuellen Maschine ohne Instrumentierung des Gastbetriebssystems zu überwachen [GHR10, GHR12].

Emulationswerkzeuge [HR02] nutzen Timer des Betriebssystems, um die Auslieferung von Rahmen zu verzögern und somit die Eigenschaften der virtuellen Verbindungen (Bandbreite und Verzögerung) zu emulieren. Die auf Interrupts basierenden Timer des Betriebssystems erfordern im Falle der Emulation von Hochgeschwindigkeitsnetzen ( $> 1 \text{ Gbps}$ ) eine hohe Interrupt-Rate. Die mit jedem Interrupt verbundenen Kontextwechsel führen zu einem erheblichen Mehraufwand. Zur Minimierung des Mehraufwands wird das Konzept der ereignisgesteuerten Timer [GMHR08] eingeführt. Hierbei lösen im Emulationswerkzeug eintreffende Rahmen das Versenden verzögter Rahmen aus. Somit kann eine hohe Interrupt-Rate und der daraus resultierende Mehraufwand vermieden werden.

In Emulationssystemen auf Basis von Knotenvirtualisierung übersteigt typischerweise die Anzahl an virtuellen Knoten die Anzahl verfügbarer physischer Netzwerkadapter. Daher wird mittels Multiplexing-Techniken eine Vielzahl virtueller Verbindungen über einen physischen Netzwerkadapter übertragen. Existierende Verfahren [Her05, Mai11] auf Basis von Hardware-gestützter VLANs (IEEE 802.1q [IEE06]) sind hierbei durch die Begrenzung von VLAN auf 4.096 virtuelle Verbindungen innerhalb der Emulationsumgebung begrenzt. Um diese Limitierung aufzuheben, wurde der VLAN-basierte Ansatz um ein Verfahren auf Basis eines softwarebasierten Multiplexing-Verfahrens erweitert. Hierbei können sich mehrere virtuelle Verbindungen ein VLAN teilen und so die Gesamtzahl emulierbarer virtueller Verbindungen erhöht werden.

Zur Verhinderung von Überlast bei gleichzeitiger Maximierung der Ausführungs geschwindigkeit von Experimenten wurde das Verfahren der adaptiven virtuellen Zeit entwickelt [GHR09a]. Die Grundidee ist hierbei, die Last der physischen Knoten beziehungsweise der virtuellen Maschinen zu überwachen und in Abhängigkeit der aktuellen Last die Ausführungs geschwindigkeit des Experiments zu erhöhen beziehungsweise zu verlangsamen. Hierbei muss die Auslastung der Emulationsumgebung auf einen Wert geregelt werden, der es bei einer Steigerung des Ressourcenbedarfs erlaubt, eine

rechtzeitige Reduzierung der Ausführungsgeschwindigkeit vorzunehmen. Im Gegensatz zu Emulationsumgebungen mit konstanter Ausführungsgeschwindigkeit kann mittels adaptiver virtueller Zeit die Laufzeit von Experimenten mit wechselndem Ressourcenbedarf maßgeblich reduziert werden.

Die Evaluation der vorgestellten Konzepte zeigt die Effizienz der entwickelten Architektur bezüglich dem Speicheraufwand und der Kommunikationskosten sowie die Genauigkeit bei der Emulation von Netzeigenschaften wie Bandbreite und Verzögerung. Ebenso zeigt die Evaluation der adaptiven virtuellen Zeit die Effektivität der Regelung der Ausführungsgeschwindigkeit von Experimenten. Einerseits belegt die Evaluation, dass es trotz wechselnder Ressourcenanforderungen zu keiner Verfälschung der Ergebnisse kommt und andererseits eine Optimierung der Ausführungsgeschwindigkeit erfolgt.

## 5 Experimentkonfiguration

In Kapitel 5 werden Verfahren zur Minimierung der Experimentlaufzeit durch Optimierung der Knotenplatzierung vorgestellt. Hierfür wird zunächst ein Überblick der Werkzeugunterstützung zur Experimentbeschreibung in NET gegeben. Im Folgenden wird ein Modell für Emulationsumgebungen entwickelt, um die Laufzeit von Netzwerkexperimenten auf Basis der Experimentbeschreibung sowie der erwarteten Datenraten zu berechnen. Dieses Modell dient als Basis für den entwickelten Platzierungsalgorithmus *NETplace* zur Berechnung einer initialen Knotenplatzierung. Um die Laufzeit von Experimenten auch bei wechselnden Ressourcenanforderungen während eines Experiments zu minimieren, wird das Verfahren *NETbalance* vorgestellt. Hierbei wird bei Ressourcenänderungen zur Experimentlaufzeit die Knotenplatzierung optimiert, um eine laufzeitoptimale Platzierung wiederherzustellen. Teile der vorgestellten Konzepte wurden bereits zuvor in Tagungsbänden [GHR10, GHR11] sowie in einem Fachblatt [GHR12] veröffentlicht.

Der Einsatz von virtuellen Protokollstapeln innerhalb der virtuellen Maschinen macht eine Unterscheidung der Kommunikationskosten virtueller Verbindungen nötig. Hierbei verursachen Verbindungen zwischen virtuellen Knoten innerhalb einer virtuellen Maschine die geringsten Kosten. Verbindungen zwischen virtuellen Knoten in verschiedenen virtuellen Maschinen innerhalb eines physischen Knoten benötigen etwa eine Größenordnung mehr Ressourcen. Sind die virtuellen Knoten einer Verbindung auf verschiedenen physischen Knoten platziert, verdoppeln sich die Kosten nochmals.

Basierend auf diesen Kostenfaktoren, der Platzierung der virtuellen Knoten, den erwarteten Datenraten der virtuellen Knoten sowie dem erwarteten Ressourcenverbrauch der zu testenden Software, kann der Ressourcenbedarf der physischen Knoten beziehungsweise der virtuellen Maschinen berechnet werden. Auf der Basis der benötigten Ressourcen sowie der Kapazität der physischen Knoten kann die Experimentlaufzeit berechnet werden.

Wegen der Schwere des Platzierungsproblems (NP-hard) wird ein auf Heuristiken basierendes Verfahren (*NETplace*) mit anschließender Optimierung zur Berechnung der initialen Knotenplatzierung vorgeschlagen [GHR10, GHR12]. Hierbei wird zunächst die virtuelle Topologie in einen gewichteten Graphen umgewandelt, wobei die Knotengewichte den erwarteten Datenraten und die Knotengewichte dem erwartenden Ressourcenbedarf der zu testenden Software entsprechen. Durch zweimalige Partitionierung dieses Graphen mittels des *k-way edge-cut* Algorithmus [KK98a] werden die Knoten zunächst auf die physischen Knoten und dann auf die virtuellen Maschinen verteilt. In einer anschließenden Optimierungsphase wird diese Platzierung durch eine auf *Hill-Climbing* basierte Optimierung verbessert.

Wechselnde Ressourcenanforderungen der virtuellen Knoten können trotz einer optimalen initialen Platzierung zu zeitweise suboptimalen Platzierungen führen und somit die Experimentlaufzeit verlängern. Durch eine Umplatzierung von virtuellen Knoten während des Experiments kann eine Platzierung hergestellt werden, welche eine Beschleunigung der virtuellen Zeit erlaubt (*NETbalance* [GHR11, GHR12]). Hierfür wird mittels einer Lastüberwachung der Ressourcenbedarf der virtuellen Knoten überwacht und an einen Koordinator gesendet. Auf Basis dieser aktualisierten Lastinformationen kann der Koordinator eine optimierte Platzierung berechnen. Durch Migration virtueller Knoten zwischen physischen Knoten kann die optimierte Platzierung in der Emulationsumgebung umgesetzt werden. Hierfür wurde ein Verfahren entwickelt, welches es erlaubt, virtuelle Knoten während eines laufenden Experiments ohne Beeinflussung der Experimentergebnisse zu migrieren. Damit die aus der optimierten Platzierung resultierende beschleunigte Experimentausführung auch zu einer Laufzeitreduktion des Experiments führt, müssen jedoch zunächst die Migrationskosten kompensiert werden. Mittels eines Kostenmodells für die Migrationskosten können diese berechnet und mit der erwarteten Laufzeitverkürzung verrechnet werden. Zur Minimierung des Migrationsaufwands wurde eine Architektur entwickelt, um Dateien mit ausschließlichem Lesezugriff (z.B. Bibliotheken der zu testenden Software) zwischen virtuellen Knoten zu teilen. Durch den gemeinsamen Zugriff auf diese Dateien müssen diese bei einer Migration nicht zwischen den virtuellen Knoten transferiert werden.

Im Zuge der Evaluation der vorgestellten Konzepte wird zunächst die Genauigkeit des Modells für Emulationssysteme untersucht. Hierbei zeigt sich, dass das entwickelte Modell die Last der Komponenten mit einer Genauigkeit von etwa 8,9 % sowie die Laufzeit der Experimente mit einer Genauigkeit von etwa 4,3 % vorhersagen kann. Bezuglich der Evaluation des Verfahrens zur initialen Platzierung zeigt sich, dass *NETplace* die Laufzeit gegenüber dem Referenzalgorithmus um bis zu 60 % reduzieren kann. Die Evaluation der dynamischen Neuplatzierung mittels *NETbalance* belegt, dass eine Migration von virtuellen Knoten während eines Experiments, transparent für die zu testende Software, durchgeführt werden kann. Zudem belegen die Evaluationsergebnisse, dass *NETbalance* die Laufzeit von Netzwerkexperimenten gegenüber einer initialen Platzierung um bis zu 70 % reduzieren kann.

## 6 Zusammenfassung

Der Softwaretest ist ein essentieller Bestandteil der Softwareentwicklung. Hierbei wird neben einer Funktionsprüfung auch die Leistungsfähigkeit der Software evaluiert. Die Zielumgebung beeinflusst maßgeblich die Leistung verteilter Software und muss daher bei der Evaluation berücksichtigt werden. Die Methode der Netzwerkemulation kombiniert die Vorteile der Netzwerksimulation und der Evaluation in Realwelt-Testumgebungen. Sie erlaubt wiederholbare Experimente von unmodifizierter verteilter Software in benutzerdefinierten Umgebungen.

Um die Nützlichkeit der Netzwerkemulation zu maximieren, liegt der Fokus dieser Arbeit auf der Entwicklung von Konzepten zur Maximierung der unterstützten Szenariengröße und zur Minimierung der Experimentlaufzeit. Hierfür wurden zunächst die zugrunde liegenden Techniken der skalierbaren Netzwerkemulation betrachtet. Neben verteilten Emulationswerkzeugen zur Emulation von Netzeigenschaften wurden die Konzepte der Knoten- und Zeitvirtualisierung diskutiert. Die Knotenvirtualisierung ermöglicht es, eine Vielzahl an Instanzen der zu testenden Software pro physischem Knoten der Emulationsumgebung auszuführen. Das Konzept der Zeitvirtualisierung erlaubt es, Experimente verlangsamt auszuführen und somit Ressourcenengpässe zu vermeiden.

Zur Minimierung der Experimentlaufzeit wurde zunächst eine Architektur entwickelt, welche effiziente Knotenvirtualisierung bei gleichzeitiger transparenter Zeitvirtualisierung unterstützt. Ereignisgesteuerte Timer erlauben hierbei eine realistische Emulation von Hochgeschwindigkeitsnetzen bei gleichzeitiger Reduzierung des Mehraufwands.

Mittels des entwickelten Konzepts der adaptiven virtuellen Zeit wird die Ausführungs geschwindigkeit des Experiments an dessen Ressourcenbedarf angepasst. Somit kann eine Reduktion der Laufzeit bei gleichzeitiger Verhinderung von Ressourcenengpässen erreicht werden.

Neben der Effizienz der Emulationsumgebung beeinflusst die Platzierung virtueller Knoten auf die physischen Knoten der Emulationsumgebung maßgeblich die Ausführungs geschwindigkeit von Experimenten. Zur Optimierung dieser Platzierung wurde zunächst ein Modell zur Berechnung der Experimentgeschwindigkeit auf Basis der Experimentbeschreibung entwickelt. Hierauf aufbauend wurde der Platzierungs algorithmus *NETplace* zur Berechnung einer laufzeitoptimalen Platzierung entwickelt. Um die Platzierung an wechselnde Ressourcenanforderungen während eines Experiment laufs anpassen zu können, wurde *NETbalance* entwickelt. Mittels *NETbalance* kann während eines Experimentlaufs eine Platzierung zur Minimierung der Restlaufzeit eines Experiments berechnet und durch Migration von virtuellen Knoten in der Emulations umgebung umgesetzt werden.

Die Evaluation belegt die Effizienz der entwickelten Emulationsarchitektur. Diese erlaubt es, tausende virtueller Knoten pro physischem Knoten auszuführen. Ferner zeigte sich, dass das Konzept der adaptiven virtuellen Zeit effektiv Überlast vermeidet und gleichzeitig die Ausführungs geschwindigkeit optimiert. Bezuglich der Leistung des Platzierungs algorithmus *NETplace* belegte die Evaluation eine Reduzierung der Experimentlaufzeit im Vergleich zu bisher genutzten Platzierungsverfahren von bis zu 60 %. Ebenso konnte gezeigt werden, dass eine Anpassung der Platzierung an sich ändernde Ressourcenanforderungen die Laufzeit von Experimenten um bis zu 70 % reduzieren kann.

Zusammenfassend stellen die in dieser Arbeit entwickelten Methoden eine große Verbesserung der Netzwerkemulation dar. Die Effizienz und Skalierbarkeit der entwickelten Konzepte erlauben die Evaluation von unmodifizierten Anwendungen und Kommunikationsprotokollen mit einer Vielzahl an Instanzen der zu testenden Software. Darüber hinaus ermöglichen es die Konzepte *NETplace* und *NETbalance*, dass Wissenschaftler mehr Experimente in kürzerer Zeit ausführen können und somit Ergebnisse von höherer statischer Relevanz erzielen können. Schlussendlich kann durch *NETbalance* der Aufwand für die Vorbereitung von Netzwerkexperimenten erheblich reduziert werden, da kein Vorwissen mehr über das Verhalten der zu testenden Software benötigt wird.

In möglichen zukünftigen Arbeiten kann die Nützlichkeit der Netzwerkemulation weiter gesteigert werden, indem Methoden entwickelt werden, um beispielsweise Netzwerk-

## Contents

---

emulatoren um die Anwendungsunterstützung von mobilen Betriebssystemen wie Android oder iOS zu erweitern. Zudem können in zukünftigen Arbeiten Methoden zur weiteren Reduzierung der Laufzeit von Netzwerkexperimenten entwickelt werden. Insbesondere bei Experimenten mit langer Initialisierungsphase kann das Anlegen von Experimentschnappschüssen genutzt werden, um diese Initialisierungsphase bei wiederholter Ausführung des Experiments zu überspringen.

# 1

# Introduction

## 1.1 Motivation

Software test is an integral part of the software engineering life cycle [Pre92]. The general goal is to check whether the software complies with requirements documented in the specification. Besides the functional requirements defined by the use cases in the specification, the performance of the software is evaluated. Performance tests are also used in research, e.g., to evaluate the behavior of newly developed distributed software (distributed applications or communication protocols). However, the performance of a distributed software heavily depends on the execution environment. A simple data transmission that delays the transmission of a frame until the reception of an acknowledgement of the previous frame, can achieve a high channel utilization on channels with small bandwidth-delay product. However, a large bandwidth-delay product results in a worse channel utilization. Therefore, accurate performance evaluations need to consider the target environment of the software to be evaluated.

In the field of distributed software, there are mainly four types of performance evaluation methodologies [Liu08a]: analytical models [Gro06], real world testbeds [CCR<sup>+</sup>03, ABKM01], network simulation [Liu08b, Fuj89, Ril03], and network emulation [AC06, HRS<sup>+</sup>08, GMHR08]. Analytical models, such as *Network Queuing Models*, model the resources (e.g., CPUs, discs, channels) of the distributed system as queues. These queues are connected according to the information flow of the system to be evaluated. Using queuing theory it is possible to evaluate several performance metrics, such as the average time a message spent in the system of a specific queue, or answer questions like, what if we increase the capacity of that link by factor 2. However, the analytical models

are typically based on the design of the software. Therefore, they cannot be used to evaluate the performance of the actual implementation. Additionally, the complexity of distributed systems often makes an analytical evaluation infeasible.

Real world testbeds [CCR<sup>+</sup>03, BL03] allow for evaluating the actual implementation of the *Software under Test* (SuT). Here, instances of the SuT are executed on computers distributed around the world. Since the Internet is used to connect these devices, the SuT is tested under realistic conditions. However, the properties of Internet-based connections, such as throughput, are not controllable by the experimenter and, thus, experiments can hardly be repeated with the same network conditions. Additionally, the number of SuT instances is limited to the number of testbed nodes.

Experiments with an arbitrary number of nodes and topologies can be performed using network simulation [Liu08b, Fuj89, Ril03]. Here, the network properties and the behavior of the SuT are simulated by software models, which allows for controllable and repeatable results. In order to achieve reasonable simulation times, these models introduce some abstractions from the real world (e.g., the operating system of the nodes is not simulated). To enable large scale experiments with millions of nodes, some simulators abstract from individual packets and simulate only packet flows [LFG<sup>+</sup>01, Kid05]. Choosing the right level of abstraction to accurately model the behavior of the SuT is a non trivial task.

Network emulation [HRS<sup>+</sup>08, GMHR08, AC06], which combines the benefits of network simulation and real world testbeds, allows for running reproducible experiments for evaluating the performance of distributed applications and communication protocols in user-defined networks without adapting the SuT to a simulation framework. These networks are modeled by connecting routers and hosts running instances of the SuT using configurable network hardware and software tools. The parameters of these network links are adjustable and include bandwidth, delay, and loss rate [Riz97, HR02]. These experiments are executed on a cluster of commodity PC-nodes which provides a cost efficient evaluation platform [AC06, MHR07]. To enable large scale experiments, *node virtualization* [JX03, WSG02, VYW<sup>+</sup>02] allows for running multiple instances of the SuT (encapsulated in so-called *virtual nodes*) on each of these PC-nodes (called *physical nodes*).

However, the number of virtual nodes per physical node is limited, as mapping too many virtual nodes to a physical node overloads the physical node. Such an overload may bias evaluation results since the SuT experiences resource shortages that do not exist in a real execution environment. Using a virtual time [CFH<sup>+</sup>80, GYM<sup>+</sup>06] that runs slower

than real time allows for reducing the system load and, therefore, allows for increasing the number of virtual nodes per physical node. For example, a real 10 Mbps network can be used to transport 100 Mbps in the emulation if the experiment is executed 10 times slower than real time. The quotient of the time used by the experiment (virtual time) and the real time is called *time dilation factor* (TDF). With this time dilation factor (TDF=10), a physical node connected to this real network can host 10 virtual nodes, each executing a software instance that consumes 10 Mbps. Existing systems [GYM<sup>+</sup>06, ELL09] with a constant TDF choose a very conservative TDF to prevent overload even when load peaks occur. Therefore, in scenarios with changing resource requirements, the system may experience considerable underload most of the time. The consequence is that the runtime of the experiment is suboptimal because, in these situations, the execution speed of the experiment could be increased without overloading the physical resources. In order to prevent overload situations and, at the same time, optimize the experiment runtime, we introduce the concept of adaptive virtual time [GMHR08]. The basic idea is to monitor the load of the physical nodes and to dynamically adjust the clock rate to the current load situation [GHR09a].

In presence of adaptive virtual time, the physical node with maximum load determines the rate of the virtual clock and, thus, the runtime of the experiments. This load is strongly affected by the placement of virtual resources of a test scenario (hosts, routers, etc.) onto the physical nodes of the testbed. Since manual placement is not feasible for large scenarios, automatic placement tools [RAL03, ZN03, LC04, CBMP04, LLXC05] have been developed. Since these tools do not consider time virtualization, they assume physical nodes with a fixed capacity and, therefore, calculate a placement without overloading the physical nodes. As a result of time virtualization, for any given placement of virtual nodes, the virtual clock rate can be adjusted such that no physical resources are overloaded. However, slowing down the virtual clock rate increases the experiment runtime. Therefore, we need a placement tool that calculates a placement of virtual nodes, that minimizes the load of the maximum loaded physical node and thus the experiment runtime. We call the placement of virtual nodes at the beginning of an experiment the *initial placement*.

Even with an optimal initial placement, varying load of virtual nodes during the experiment may result in a temporarily suboptimal placement and, thus, in a suboptimal execution speed of the experiment. By monitoring the load of the virtual and physical nodes, we can detect these situations and calculate an optimized placement. Using the migration of virtual nodes, the optimized placement can be established in the testbed. This migration allows for balancing the load between the physical nodes and, thus,

avoids a high load on single nodes, which is the main reason for a suboptimal experiment runtime. In combination with adaptive virtual time, the migration of virtual nodes allows for minimizing the runtime of network experiments with varying load.

## 1.2 Contributions

In order to minimize the runtime of network emulation experiments, this thesis makes the following contributions:

- A classification of existing performance evaluation tools based on the three building blocks of scalable network emulation: distributed emulation tool, node virtualization and time virtualization. (cf. Chapter 2)
- A lightweight node virtualization approach supporting multi-core architectures to provide virtual time transparently to the *Software under Test* with low memory and communication overhead. (cf. Section 4.1)
- A mechanism to ensure the accuracy of frame delays in a network emulation tool despite time virtualization. (cf. Section 4.2)
- An efficient multiplexing scheme for virtual links to abolish the hardware limitations of VLAN-based emulation approaches. (cf. Section 4.3)
- An epoch-based time virtualization to minimize the experiment runtime by maximizing the hardware utilization during experiments with minimum synchronization overhead. (cf. Section 4.4)
- A generic cost model for the communication between virtual nodes to estimate the runtime of network emulation experiments. (cf. Section 5.2)
- An algorithm to calculate an initial placement of virtual nodes onto physical nodes that minimizes the runtime of network experiments. (cf. Section 5.3)
- An architecture to efficiently support virtual node migration during the experiment execution without altering the emulation results. (cf. Section 5.4.2)
- An approach to minimize the experiment runtime by adapting the placement of virtual nodes to varying resource requirements of the virtual nodes based on transparent node migration. (cf. Section 5.4.4)
- An extensive evaluation showing the accuracy and scalability of our network emulator as well as the effectiveness of our approaches to minimize the experiment runtime. (cf. Section 4.5 and Section 5.5)

## 1.3 Structure

The remainder of this thesis is structured as follows: In Chapter 2, we first introduce the three building blocks of scalable network emulation: distributed network emulation tools, node virtualization, and time virtualization. Second, we classify existing approaches for evaluating the performance of distributed applications and communication protocols.

In Chapter 3, we first introduce the *Network Emulation Testbed* which acts as a foundation for this work. Second, we present the general architecture and the interaction of the components discussed in this thesis. We conclude the chapter by a discussion of application case studies using our emulation testbed for performance evaluations.

Chapter 4 introduces the concepts for efficient node and time virtualization. These concepts include an architecture for efficient node and time virtualization with low overhead, a mechanism to multiplex virtual links on a single physical network interface, and an approach to adapt the virtual clock rate to the system load. A detailed evaluation investigates the introduced overhead of the presented approaches as well as the achieved level of resource utilization using the adaptive virtual time concept. A summary concludes this chapter.

The experiment configuration is discussed in Chapter 5. After introducing the experiment workflow, a detailed cost model of an emulation testbed followed by approaches to specify an experiment are discussed. Based on this testbed model, we introduce an algorithm (*NETplace*) to calculate an initial placement of virtual nodes onto physical nodes that minimizes the experiment runtime. Using *NETplace* as a foundation, we present an extended approach, called *NETbalance*, to adapt the initial node placement during the experiment runtime to cope with changing resource requirements of virtual nodes. Detailed evaluations show the improvements in terms of reduced experiment runtime of the presented concepts. The chapter concludes with a summary.

The thesis is closed by Chapter 6. Here, we give a summary of the contributions of the thesis and draw conclusions. Finally, we discuss promising future research directions.



# 2

# Background

This chapter gives an overview on the three basic building blocks of scalable network emulation: First, a mechanism to emulate link properties such as bandwidth, delay and packet loss. Second, *node virtualization* to run multiple instances of the *Software under Test* (SuT) on each physical node. Third, *time virtualization* to avoid overloading the physical resources. Based on that, we classify the related work in the field of performance evaluation of distributed systems.

## 2.1 Emulation of Link Characteristics

In real world testbeds [CCR<sup>+</sup>03, BL03] link properties result from the location of the hosts running the SuT in the underlying network (e.g., the Internet). Moreover, the current traffic situation in the Internet influences the properties of the links. Additionally, the testable topologies are limited to the topology of the testbed or a subset of it.

Network Emulation allows for evaluating the SuT using arbitrary topologies with user-defined link characteristics [GHR09b]. The links between the instances of the SuT can be modeled using two different approaches: a distributed network emulation tool [HR02, Riz97] or an attached network simulator such as NS [Fal99], IP-TNE [SU03], OMNeT++ [TRR08, WSHW08], and PRIME [LLH09].

In case of a distributed network emulation tool the network topology and the link characteristics are modeled by two different mechanisms. Initially, the emulated network topology is modeled by physically connecting the network interface cards of the physical nodes. Techniques such as VLAN (IEEE 802.1q [IEE06], (cf. Section 4.3) and node virtualization [MHR07] (cf. Section 2.2) replace the physical wires by virtual counterparts.

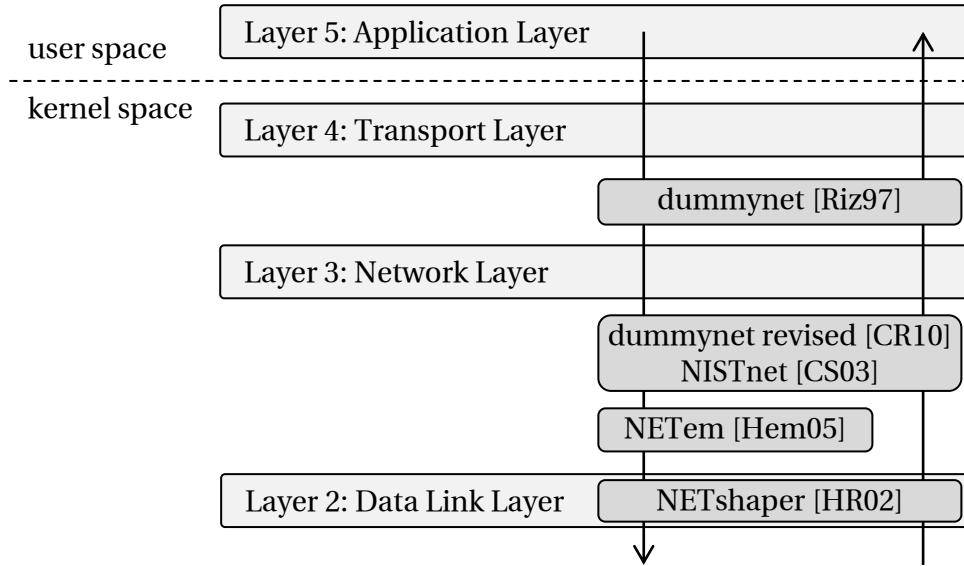


Figure 2.1: Protocol stack extended by emulation tools

The emulation tool itself only emulates the characteristics of the links such as delay, bandwidth and loss. Therefore, the emulation tool is inserted into the protocol stack and is attached to each instance of the SuT. Frames to and from the SuT are piped through the emulation tool and, thus, the tool can delay and drop frames to emulate the link characteristics.

The emulation tool can be inserted at different locations into the protocol stack to intercept frames (cf. Figure 2.1). *NETshaper* [HR02, GMHR08] acts as a virtual Ethernet device on Layer 2 and, thus, it allows for evaluating protocols and applications on Layer 3 and above. Inside *NETshaper* a queue is used to delay frames. *NETem* [Hem05] intercepts outgoing packets at the interface between Layer 2 and 3 by acting as a Linux queuing discipline and, thus, only allows for delaying or dropping outgoing packets. *NISTnet* [CS03] and a revised version of *dummmynet* [CR10] act as packet classifiers to intercept packets at the Layer 2–Layer 3 interface. Intercepted packets are matched against a rule table and matching rules are executed. These rules allow for specifying the delay, the bandwidth and the loss of the links between instances of the SuT. Finally, the original *dummmynet* [Riz97] uses hooks at the Layer 3–Layer 4 interface to evaluate distributed applications (Application Layer) and protocols on the Transport Layer.

An alternative to the application of emulation tools is network emulation using an attached network simulator. Based on DES (*Discrete Event Simulation*) [Fis78], the network topology and the link characteristics are simulated. The SuT is not part of the simulated network model, moreover, unmodified applications or communication pro-

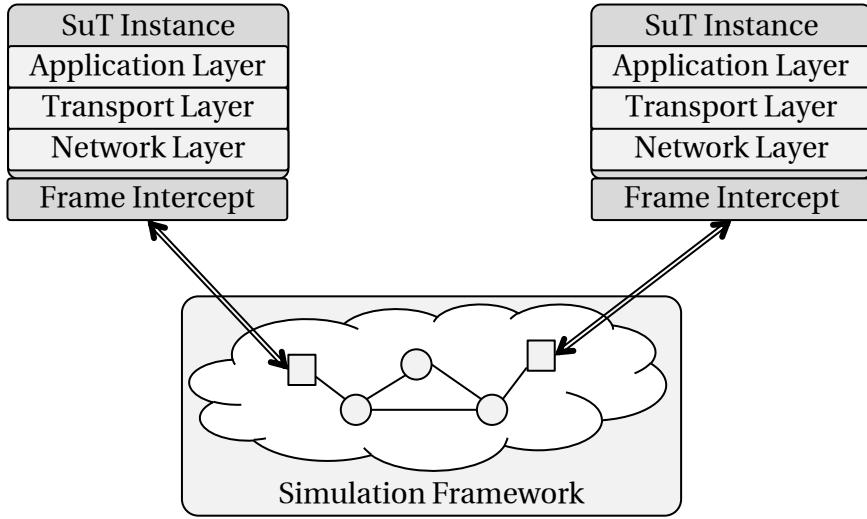


Figure 2.2: Network emulation using simulation frameworks

tocols are executed. As shown in Figure 2.2, the network traffic of the SuT is intercepted and forwarded to the simulation framework. Most centralized network simulators [WVLW09], such as NS [Fal99], NS-2 Emulation [KP09], NS-3 [AOC<sup>+</sup>10], and OMNET++ [TRR08], support the interception and the processing of real network traffic within the simulation. The usage of a parallel simulation framework, based on PDES (*Parallel Discrete Event Simulation*) [SU03, LLH09], provides enough resources to keep the execution of a simulation model, even for large network topologies, synchronized with the SuT.

## 2.2 Node Virtualization

Running only one instance of the *Software under Test* (SuT) on each physical node of the emulation testbed limits the scalability of network emulation. To get around this limitation, the concept of node virtualization allows for partitioning of the physical testbed resources and, thus, allows for running multiple instances of the SuT on each physical node. Each partition is called *virtual node*. These virtual nodes are connected by the emulation tool (cf. Section 2.1) and run an instance of the SuT.

Resource partitioning can be done on different layers. The spectrum ranges from emulating the entire hardware platform [Boc12, Bel05] including the emulation of the processor architecture to the separation of process memory [PF07, KW00], where each process runs in exclusive virtual memory as provided by common operating systems. Here, we focus on concepts that are typically used for network emulation and that support partitioning of the required resources: virtual machines and virtual protocol stacks. Figure 2.3 shows the resource virtualization approaches classified by means of flexibility and efficiency. In the following sections, we discuss the different approaches for virtualization based on virtual machines and virtual protocol stacks in detail.

### 2.2.1 Virtual Machines

Virtual machines (VMs) [SM79, Cre81] are used to partition the physical resources of a computer by virtualizing the hardware. The software running inside the VM can access the virtual hardware in the same way as if it directly accesses the real hardware.

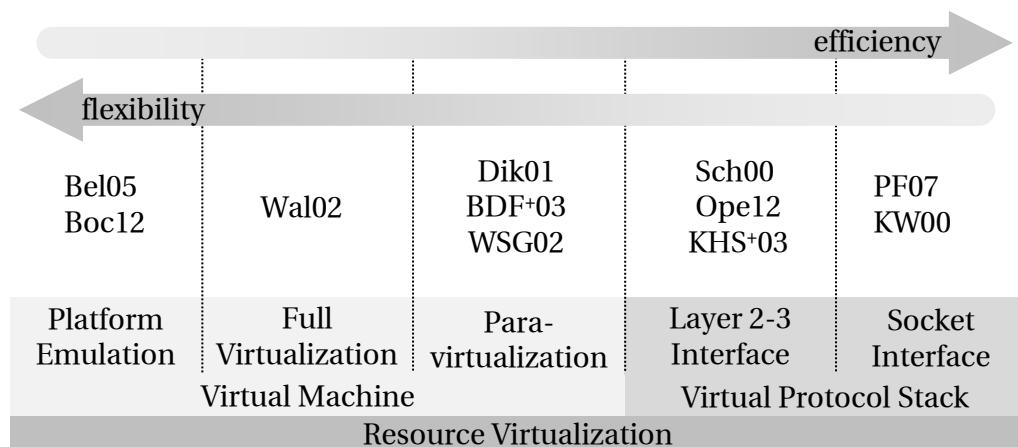


Figure 2.3: Resource virtualization approaches

Therefore this virtualization is transparent to software running inside the VM. Since multiple VMs share the same physical resources, a software called *Virtual Machine Monitor* or *Hypervisor* is used to schedule and manage the access of the VMs to these resources. The hypervisor ensures that the software instances, running in different VMs, are isolated from each other, and cannot access or modify resources of other VMs.

Emulation approaches based on virtual machines execute each virtual node inside a virtual machine [AC06]. The protocol stacks are located on top of virtual network devices, which are connected to other VMs by a software switch with an uplink to other computers. Due to the virtualization at the hardware interface, such approaches are fully transparent to the SuT.

Virtual machines can be classified by means of the provided hardware abstraction. Approaches such as BOCHS [Boc12] or QEmu [Bel05] emulate an entire hardware platform including the processors instruction set architecture. Therefore, these approaches can be used to evaluate software written for specific processor architectures, e.g., operating systems for mobile devices running on MIPS [PH08] or ARM [Fur00] processors. However, providing a processor architecture differing from the processor architecture of the testbed introduces a high overhead, because every instruction of the virtual CPU must be translated and executed by a set of instructions on the physical CPU.

Providing a virtual CPU with a processor architecture equal to the processor architecture of the physical CPU reduces the computation overhead by an order of magnitude [GWS06]. Using CPU virtualization [Gol74], instructions of the virtual CPU can be directly executed on the physical CPU. Privilege levels of the CPUs are used to provide resource isolation. However, Goldberg [Gol73] shows that the used privilege level changes the result of some instructions, e.g., virtual memory access [Ros04] of the x86 processor architecture [Int10]. Rose [Ros04] calls these non-virtualizable instructions "problem instructions". In literature, there are two approaches to virtualize those instructions: *full virtualization* and *paravirtualization*.

Full virtualization [Wal02] uses just in time code translation to replace the non-virtualizable instructions. After the replacement the instructions of the virtual CPU can be executed directly on the physical CPU. However, the replacement introduces runtime overhead [Ros04]. Systems based on paravirtualization, such as Xen [BDF<sup>+</sup>03] or UML [Dik01], avoid this overhead by modifying the guest operating system such that these non-virtualizable instructions are not used. In addition, the *Virtual Machine Monitor* or *Hypervisor* provides only a simplified hardware which can be accessed by the guest operating system efficiently [FHN<sup>+</sup>04b].

Each virtual machine runs a guest operating system which causes memory overhead. The same buffer cache entries may possibly exist in each guest and they may exist in the host operating system or virtual machine monitor, too. A minimal Linux instance needs several megabytes of memory when executed in a VM based on the Xen [BDF<sup>+</sup>03] hypervisor. Equal program libraries used in different VMs at the same time increase the memory overhead. While the memory overhead can be mitigated by content based page sharing [GLV<sup>+</sup>08, VMC<sup>+</sup>05, Wal02], this in turn implies some computation overhead to calculate and compare page content. Lightweight VMs, such as the Denali Isolation Kernel [WSG02, WSG04], combines a simplified hardware interface with a custom, size-optimized guest operating system to minimize the memory overhead.

Network communication between virtual nodes based on VMs requires expensive context switches involving the hypervisor. Approaches like *XenLoop* [WWG08] avoid copying memory pages between the VMs and the hypervisor using shared memory-based channels between the VMs. Nevertheless, the overhead of virtual machines reduces the possible scenario sizes significantly [BQ06a, MGWR07, CGMV07, SPF<sup>+</sup>07]. The overhead of virtual machines can be partially reduced by offloading functionality of the hypervisor to the hardware [RGSX06, BYMX<sup>+</sup>06]. However, the overhead introduced by virtual machines can be further reduced by the concept of *virtual protocol stacks* (discussed in the following section). In previous work [MGWR07], we were able to execute six VMs on a physical node, while on the same hardware, a lightweight virtualization with *virtual protocol stacks* allowed for running scenarios with up to 30 virtual nodes.

### 2.2.2 Virtual Protocol Stacks

Partitioning of the operating system allows for minimizing the node virtualization overhead [Mai11]. All virtual nodes running on a physical node share a common operating system. Using protocol stack virtualization [KHS<sup>+</sup>03] each virtual node can be attached to a separated instance of the protocol stack.

Figure 2.4 shows two possible approaches to virtualize the protocol stack. Using virtualization at the Layer 2-3 interface [KHS<sup>+</sup>03] (cf. Figure 2.4a), each virtual node has its own protocol stack, including own routing tables, on top of a virtual network device. In contrast, all virtual nodes share the network and transport layer state in case of a virtualization at the socket interface [PF07] (cf. Figure 2.4b). However, as a benefit of the latter approach, it can be implemented entirely in the user space by intercepting the system calls of the *Software under Test*. Virtualization of additional operating system

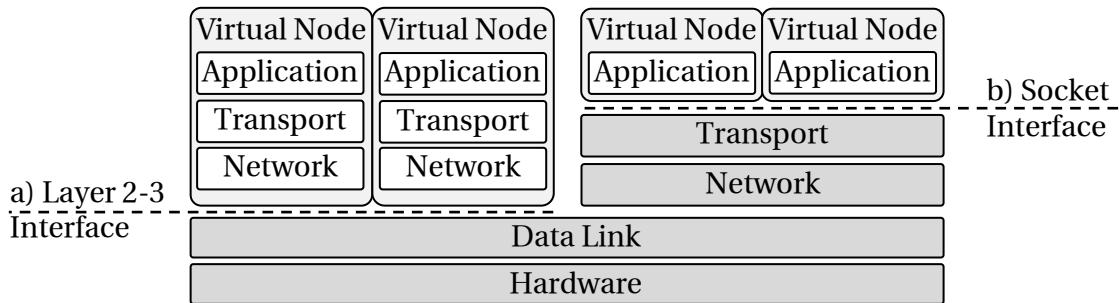


Figure 2.4: Virtual nodes based on protocol stack virtualization

resources like name spaces and file systems [BDM99, Sch00, KW00] allows for isolating virtual nodes and, therefore, are completely transparent to the SuT.

The benefit of virtual protocol stacks is that only one operating system runs on each physical node, resulting in a significantly reduced memory and computation overhead [BQ06b]. Since all SuT executed on the same physical node run under a single operating system, program libraries and operating system caches can be shared between the virtual nodes. Due to the shared operating system, virtual nodes running on the same physical node can communicate using reference passing without additional, expensive context switches.

The comparison of virtual machine-based and virtual protocol stack-based emulation approaches shows that virtual protocol stacks imply less overhead and, thus, provide higher scalability. The only remaining limiting factors for emulating large scale scenarios with thousands of virtual nodes per physical node are the computation and network resources provided by the testbed hardware. These limitations can be abolished by the concept *time virtualization*, which we discuss in the next section.

## 2.3 Time Virtualization

Emulation systems [HRS<sup>+</sup>08, AC06, Mai11] typically execute experiments at real time. This approach requires at any time enough resources to run the SuT and the emulation tools. A resource shortage or resource overload can bias the emulation results, because for example frames may experience additional undesired delay.

Node virtualization (cf. Section 2.2) increases the resource consumption of physical nodes and, therefore, increases the risk of resource overload. The state of the art solution is to monitor the resource consumption to detect overload [HRS<sup>+</sup>08, Mai11] and, in case

of an overload situation, the experiment is repeated with less virtual nodes per physical node until the experiment can be completed without overload. However, executing an experiment multiple times increases the overall experiment runtime.

Instead of repeating the experiment with fewer virtual nodes on each physical node, Gupta et al. [GYM<sup>+</sup>06] adopted the concept of virtual time [CFH<sup>+</sup>80] to network emulation. The basic idea is to run an experiment a factor slower than real time. The reduced execution speed of the experiment reduces the system load by artificially increasing the resources of the testbed. For example, a real 10 Mbps network can be used to transport 100 Mbps in the emulation if the *time dilation factor*<sup>1</sup> is set to 10 (virtual time runs 10 times slower than real time). Similarly to the network load, slowing down the virtual time reduces the CPU load of the physical nodes. Therefore, virtual time allows for increasing the number of virtual nodes per physical node.

Virtual time can also run faster than real time [CWdO<sup>+</sup>06]. Thus, it can be used to discover the behavior of software with low resource consumption over a long period of time, e.g., the future behavior of malware (e.g., computer viruses and Internet worms). Therefore, the software (malware) is deployed on virtual nodes and the timely behavior of the SuT is monitored. This concept applied to network emulation could possibly reduce the runtime of network experiments as well. However, the delay ( $\ll 1$  ms) of the physical network of a network emulation testbed increases with the speed of the virtual time. While this delay has insignificant effects on the emulation results running at real time [Her05], the effect increases with speeding up the virtual time. Finally, this delay will bias the emulation results. Therefore, emulation systems only slow down the virtual time to prevent resource overload.

Several approaches increase scalability of network emulation by replacing real time with virtual time. The following sections cover different approaches for implementing and presenting virtual time.

---

<sup>1</sup>Factor describing the quotient of the experiment speed and the real time speed.

### 2.3.1 Implementation Approaches for Virtual Time

There are basically three approaches to provide virtual time to the *Software under Test*: virtual time based on virtual machines, virtual time at the system call interface, and virtual time through operating system modifications.

Gupta et al. [GYM<sup>+</sup>06] introduce the virtual time concept using virtual machines [BDF<sup>+</sup>03]. By defining a *time dilation factor* (TDF), which is used to scale the time provided by the hypervisor to the virtual machine, it is possible to run a VM with an arbitrary virtual time. All components running inside the VM, including the guest operating systems as well as the SuT experience virtual time instead of real time. Due to the virtual machine abstraction, no modifications to the SuT are required.

As discussed in Section 2.2, virtual machines introduce additional overhead. Implementing virtual time at the interface between the applications and the operating system, denoted system call interface, avoids this overhead [ZN11]. However, the lower layers of the protocol stack, including the *Network* and *Transport* layer, are typically implemented as part of the operating system and, thus, do not experience virtual time if this approach is used. Therefore, by providing virtual time at the system call interface only SuT at the *Application* layer [BF11] can be evaluated.

A third approach to provide virtual time introduces a real time independent virtual time into the operating system kernel [WK02]. The entire operating system is modified, such that the applications, the protocol stack, as well as their timers use virtual time as the time source. Using this approach, no VMs are required which results in less overhead. The missing VM abstraction causes an increased implementation overhead, because the virtual time concept has to be implemented throughout the entire kernel including the implementation of the protocol stack and not only at the interface between hypervisor and the VM. These modifications have to be applied to the SuT located at lower layers of the protocol stack, too.

Comparing the three approaches, only a virtual time implementation based on virtual machines allows for evaluation of unmodified distributed applications and communication protocols. However, this transparency comes with an increased runtime overhead.

### 2.3.2 Approaches to Virtual Time Representation

Two extreme cases for the measurement and advancement of virtual time exist in the literature: slowing down the real time by a factor or discrete events.

The *time dilation factor* (TDF) introduced by Gupta et al. [GYM<sup>+</sup>06] scales real time by a factor. This factor is constant during the entire experiment. First, such an approach leads to the problem of selecting an adequate value for the entire experiment duration. Selecting the TDF a too high wastes processing power and setting it too low results in biased emulation results. Second, the load generated by the scenario varies over time. Hence, the TDF has to be selected for periods with maximum load and, therefore, hardware resource are not optimally utilized in periods of low resource consumption.

The authors of dONE (distributed open network emulator) [BVB06] follow another approach. The idea is similar to discrete event simulation. Virtual time is advanced after processing events such as transmission of packets or triggered by timers. Such an approach provides the emulator with unlimited processing power. dONE introduces a *time warp* operator to advance virtual time and skip periods where no events happen.

In case of a parallel discrete event-based approach, synchronization mechanisms are required to schedule events without causality errors [Fuj89]. A causality error occurs when node 1 and node 2 have concurrently processed all events with timestamps less than  $t_1$  and  $t_2$ , respectively, where  $t_2 > t_1$ . If node 1 now sends a message to node 2 which has to arrive before  $t_2$  the causality error occurs because node 2 has processed at least one event that has to be executed after the arrival of the message from node 1. Two basic approaches exist to handle this problem: a conservative [CM79] and an optimistic [Jef85, JS82] synchronization scheme. The conservative scheme prevents the existence of causality errors by executing events only if it can be guaranteed that no other node sends a message that has to be processed before. In case of an optimistic scheme, events can be executed without limitations, but when a causality error is detected, the simulation state has to be rolled back.

When executing real applications, rollback-based approaches are very expensive since each memory operation of the SuT needs to be logged and possibly recovered on a rollback operation. In the field of network emulation with event-based virtual time, only conservative approaches are feasible. Since the transmission times of frames are small, this results in a huge synchronization overhead [GMHR08]. Weingärtner et al. reduce this overhead by trading the synchronization precision against the synchronization overhead while maintaining the emulation accuracy [WSHW08].

## 2.4 Performance Evaluation Tools

This section gives an overview on tools for performance evaluation of distributed systems. There are three types of tools: real world testbeds, emulation-based testbeds and network simulators. A classification of these approaches used to be simple. Real world testbeds use existing networks to evaluate the *Software under Test*, emulation-based testbeds make use of an emulation tool to build a virtual network with arbitrary network characteristics and network simulators use abstract models of the network and the SuT. Additionally, both testbeds run at real time whereas simulations are executed in virtual time.

However, in recent years the border between simulators and testbeds diffuses. Real world testbeds evolve to emulators by applying the concept of network virtualization to form arbitrary overlay networks on top of the real world networks. Emulation-based systems apply the concept of time virtualization to run the experiment with virtual time independent of real time and, therefore, provide a feature that used to be unique to network simulation. At the same time network simulators have been extended by a real time scheduler and the support for the evaluation of unmodified real applications and, therefore, emerge towards a network emulator. In the following, we discuss the recent developments of these three performance evaluation methods in detail.

### 2.4.1 Real World Testbeds

Real world testbeds, such as PlanetLab [PACR03, CCR<sup>+</sup>03] and RON [ABKM01], are closely related to network virtualization [CB10b]. Network virtualization allows for building virtual networks as a subset of a physical network [EGH<sup>+</sup>10]. These networks can be used to implement isolated experimental networks for new communication protocols and applications.

The network virtualization can be implemented on different layers. XBone [TH98] creates overlay networks on the *Network* layer. Resources within the virtual networks are addressed by IP addresses, which can be global addresses of the underlaying Internet or local addresses of the overlay. In both cases only resources that are part of the overlay can be accessed. VNET [SD04] and VINI [BFH<sup>+</sup>06] virtualize the network on Layer 2. Here, the SuT is executed in virtual machines running on arbitrary nodes with Internet access. The virtual network between these VMs is built of virtual links that are tunneled through the Internet.

In real world testbeds [PACR03, CCR<sup>+</sup>03, ABKM01] based on overlay networks, the characteristics of the virtual links are a result of the path conditions in the underlay network. Since the testbed nodes are typically operated by large research institutions, the nodes have well-connected Internet access. In order to support more heterogeneity, SatelliteLab [DHB<sup>+</sup>08] allows for connecting additional nodes, such as desktops and handhelds, via low bandwidth connections (DSL, cellular links) to the virtual network.

### 2.4.2 Network Simulation

The scalability of network simulation is increased by the concepts of parallelization and abstractions. The concept of parallelization reduces load on single computers by distributing the network model to multiple computers [Mis86, RFA99]. The computers, each simulating only a fraction of the network model, are exchanging synchronization messages to keep the distributed fractions of the network model synchronized. The performance of the distributed simulation is primarily related to the efficiency of the synchronization mechanism [Fuj89].

The concept of abstractions is used to reduce the simulation load [LFG<sup>+</sup>01, Kid05]. Instead of simulating every packet flowing through a network link, only the flow behavior of a transport layer protocol is simulated. An extension of IP-TNE [KSU05] allows mixing of packet and fluid flows to increase the scalability of the simulation. The fluid flows are used to generate synthetic background traffic on the network used to evaluate the SuT.

An opposite approach to abstraction is the introduction of emulation capabilities to network simulation. Here, the network simulator is extended by a real time scheduler and a facility to process real network traffic generated by a SuT running on separated machines or virtual machines. Such extensions are available for almost all common network simulators: NS [Fal99], NS-2 Emulation [KP09], NS-3 [AOC<sup>+</sup>10], OMNET++ [TRR08, WSHW08], IP-TNE [SU03], and PRIME [LLH09].

The concept of time dilation [CFH<sup>+</sup>80, GYM<sup>+</sup>06] can also be used with simulations extended by emulation capabilities [LR09, ELL09]. For SuT on the application level the operating system interface can be adapted to provide virtual time [WK02, BVB06]. Evaluation of SuT on the *Network, Transport* and *Application* layer without any modification of the SuT can be achieved by synchronizing a virtual machine to the simulation [WSvL<sup>+</sup>11]. Any software running inside the virtual machine experiences virtual time.

### 2.4.3 Emulation-based Testbeds

In order to evaluate unmodified distributed applications and communication protocols, distributed emulation testbeds [HR02, VYW<sup>+</sup>02, JNVP06] are used. Here, each physical node of the testbed runs one instance of the *Software under Test* (SuT). In contrast to network simulation, the experiments based on emulation run at real time.

The acquisition and operation of such testbeds is rather costly and, therefore, with the availability of virtualization technologies (cf. Section 2.2), multiple instances of the SuT can be executed on each physical node. This allows for running many network experiments on a single desktop PC. Approaches like Netkit [PR08], vBET [JX03], and VNUML [GFR<sup>+</sup>04] make use of the virtual machine UML [SPYH03] to provide virtual nodes. Experiments are restricted to a single physical node, however, the virtual topology can be connected to the Internet. VITT [CFDL08] provides the same functionality based on the more efficient virtual machine KVM [KKL<sup>+</sup>07].

The available resources of a single node limit the possible experiment size. Therefore, the resources of an emulation testbed consisting of multiple physical nodes are combined with node virtualization based on the virtual machine XEN [BDF<sup>+</sup>03] (V-eM [AC06], viNEX [MvdPJ09], and Neptune [DGCV09]). Since virtual machines come with some overhead, limiting the scalability of network emulation, a more lightweight virtualization based on virtual protocol stacks is applied in NET [MHR07] and Emulab [HRS<sup>+</sup>08].

The emulation of large networks can easily have a resource consumption that exceeds the resources of emulation testbeds. Testbeds such as Diecast [GVV08] make use of the concept of time dilation [CFH<sup>+</sup>80, GYM<sup>+</sup>06] (cf. Section 2.3) to run network experiments a factor slower than real time and, thus, virtually increase the available resources.

In this work [GMHR08, GHR09a, GHR12], we extend the lightweight virtualization approach by the concept of time dilation to increase the scalability of network emulation further. Therefore, in the following chapters, we will introduce a architecture to combine transparent virtual time based on virtual machines with efficient node virtualization based on virtual protocol stacks.



# 3

# System Overview

In this chapter, we first give an overview on the network emulation testbed used as a foundation for this thesis. Second, we briefly introduce the general architecture of our system and the interaction between the components described in Chapter 4 and Chapter 5. This chapter is closed by an introduction of research projects using our prototype implementation as a performance evaluation platform.

## 3.1 Network Emulation Testbed

The research has been conducted within the NET project (*Network Emulation Testbed*) at the Department of Distributed Systems of the University of Stuttgart. The project was funded by the German Science Foundation<sup>2</sup>. Two preceding theses [Her05, Mai11] originating from this project build the foundation of this work. In the thesis of Herrscher [Her05], a distributed emulation tool [HR02] for accurate and scalable emulation of wireless and wired networks was developed. The thesis of Maier [Mai11] extends the *Network Emulation Testbed* by a lightweight approach for node virtualization [MHR07] based on virtual protocol stacks (cf. Section 2.2).

The foundation of both theses was the emulation testbed based on a cluster of 64 commodity PC-nodes. The architecture of the cluster is shown in Figure 3.1. The physical nodes of the testbed are equipped with a 2.4 GHz Intel Pentium 4 processor, 512 MB of main memory (RAM), and two network interface cards. One 100 Mbps network interface is connected to a control network used to setup the network experiments (e.g., deploy and run the *Software under Test*). The second network interface runs at 1 Gbps

<sup>2</sup>Funded under grant DFG-GZ RO 1086/9-1-3

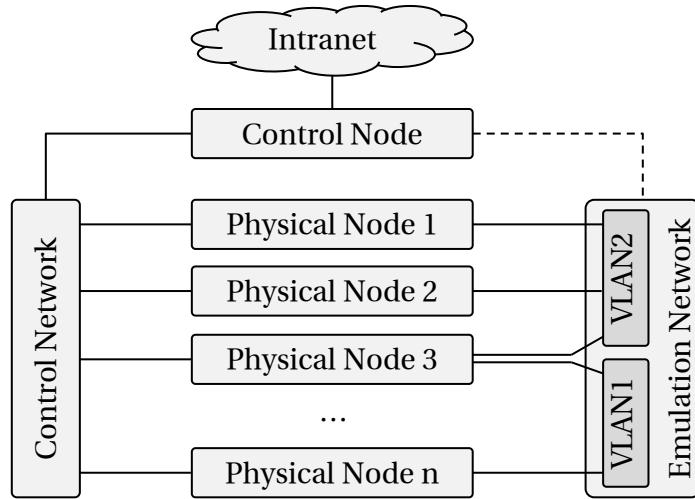


Figure 3.1: Physical architecture of the *Network Emulation Testbed*

and is connected to the emulation network. The emulation network is partitioned using VLANs (IEEE 802.1q [IEE06]) to emulate arbitrary network topologies. Details on the VLAN-based network emulation are discussed in Section 4.3.1. The emulation testbed is accessed and controlled by a dedicated *control node*. The control node is connected to the control network as well as to the intranet of the Distributed Systems Lab. The control node is also used to configure the emulation network. Further details on the hardware can be found in the thesis of Herrscher [Her05].

During the research reported in this thesis, the hardware of the network emulation testbed was replaced. The new testbed is also composed of two dedicated networks for control and emulation purposes. However, the cluster is now built of 16 PC nodes, each equipped with two QuadCore Intel Xeon processors running at 2.4 GHz and 16 GB of main memory (RAM). The evaluations presented in this thesis are partially executed on the old and on the new testbed hardware. We will refer to the used hardware at the corresponding evaluation sections.

## 3.2 General Architecture

This section gives an overview of our architecture to minimize the runtime of scalable network emulation. We briefly introduce the components of our network emulator and the interaction between them (cf. Figure 3.2). The components of our network emulator are executed on three types of devices. First, a *client* is used to specify the network experiment. Second, a *coordinator* is used to setup and to control the experiment. Third, the experiment itself is executed on a set of *physical nodes*. In the following, we discuss these components in detail.

At the beginning of a network experiment using the *Network Emulation Testbed* (NET), we first need to specify the scenario. The scenario description consists of three components:

- The *Software under Test* (SuT) running on the virtual nodes. The SuT can be a distributed application or a communication protocol.
- The network topology defining the links between the virtual nodes. These links can be parameterized by link properties such as bandwidth, delay and loss rate.

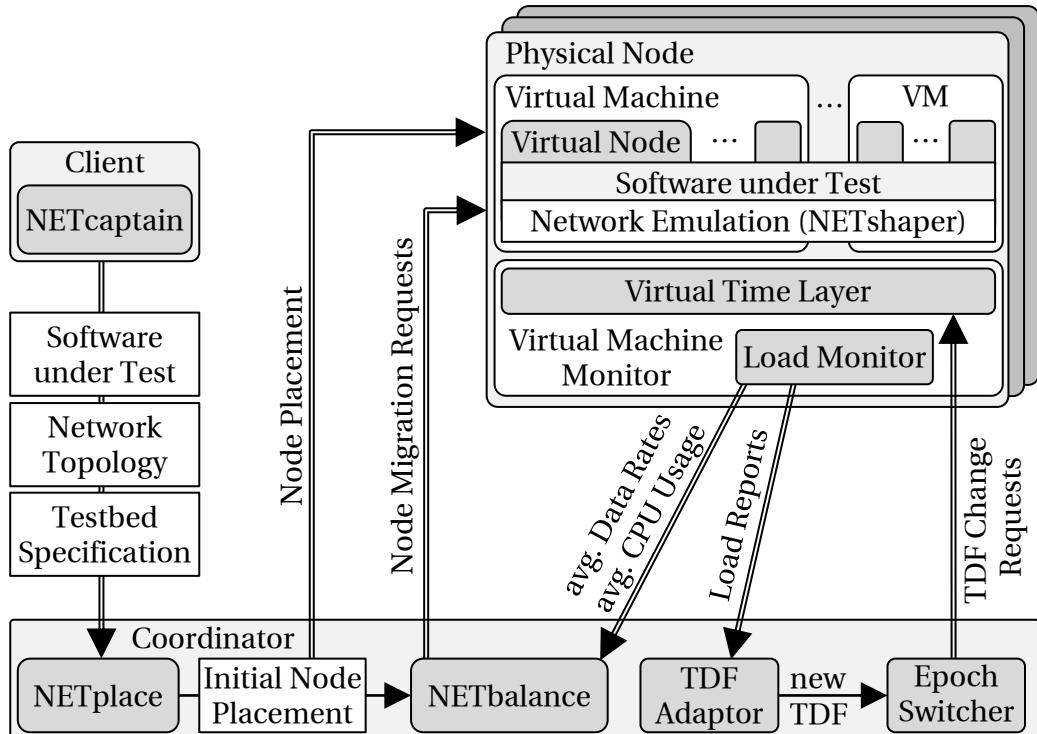


Figure 3.2: Components of the *Network Emulation Testbed*

- The testbed specification defining the number of physical nodes used to run the experiment and testbed specific parameters. The parameters include a testbed model (cf. Section 5.2) defining the load generated by transferring data between virtual nodes.

In NET, there are two alternatives to support the description of the scenario: a framework [Gra11] based on Ruby<sup>3</sup> allowing for a text-based specification of the scenario and *NETcaptain*, a GUI-based client that allows for a graphical description of the scenario. The experiment specification is discussed in detail in Section 5.1.

Taking the scenario description as an input, our placement tool *NETplace* [GHR10] (cf. Section 5.3), running on the coordinator, calculates an initial assignment of the virtual nodes onto the physical nodes to minimize the runtime of the experiment. Using the calculated placement, the scenario is deployed to the physical nodes.

In order to allow for large scale experiments with low runtime overhead, we have developed an architecture [GMHR08, GHR10] (cf. Section 4.1) to efficiently combine node and time virtualization. The network between the virtual nodes running the SuT is emulated using our distributed emulation tool *NETshaper* [HR02]. We extended the emulation approach by a scalable link multiplexing scheme (cf. Section 4.3), to overcome the limited number of virtual links supported by VLAN-based emulation.

With the large number of virtual nodes running on each physical node, the load of a physical node may exceed the capacity of the physical node. This overload can result in biased emulation results. To prevent these overload situations, we have developed the concept of *adaptive virtual time* [GHR09a] (cf. Section 4.4). Using load reports from physical nodes, the coordinator can adapt the execution speed of the experiment to the system load (cf. *TDF Adaptor* in Figure 3.2) and, therefore, effectively prevent overload of the physical node. An *epoch switcher* is used to establish the new experiment speed simultaneously on all physical nodes.

Finally, we introduced a mechanism, called *NETbalance* [GHR11] (cf. Section 5.4), to adapt the placement of virtual nodes to changed load distributions between physical nodes. Again, we monitor the load generated by the virtual nodes. Based on these load reports, we can calculate an optimized placement of the virtual nodes while considering the time to change the placement, too. The improved placement is established by migrating virtual nodes between physical nodes.

---

<sup>3</sup>Ruby scripting language: <http://www.ruby-lang.org>

### 3.3 Application Case Studies

In this section, we provide a selection of application case studies that use the *Network Emulation Testbed* for performance evaluations. These applications show the applicability and utility of our system for performance evaluation. The evaluated distributed systems are in the field of distributed pervasive applications [SHR10], distributed stream processing [RDR11], and distributed file systems [Kie09].

The first application is from the field of distributed pervasive applications [SHR10]. Here, applications are composed by several tasks (e.g. input, processing, and output) that can be deployed and run on different devices. In pervasive environments, multiple devices can provide the functionality of a task. For example a video projector or the video monitor can be used for the *output* task. In order to run an application, a configuration that binds the tasks to devices has to be calculated. With an increasing number of devices and tasks, this calculation is a complex problem. Schuhmann et al. [SHR10] have developed a distributed algorithm to run this calculation on resource-rich devices to minimize the configuration time.

During the performance evaluation of their approach, the *Network Emulation Testbed* was used to extend real-world experiments based on smart devices such as smart phones, PCs, and notebooks. Due to the limited number of physical devices, real-world experiments were performed with up to 10 devices. The emulation testbed allows for conducting scalability experiments using the same implementation as used for the real-world experiments. In these experiments up to 85 emulated devices were executed in the emulation testbed. As a conclusion, the usage of the emulation testbed enables the scalability analysis and, therefore, improves the value of the evaluation to a great extent.

The second application focuses on distributed stream processing [RDR11]. Here, sensors deployed in a network (e.g., the Internet) provide a continuous stream of sensor values. Operators are used to combine these sensor values to produce higher level information (e.g., pictures of multiple cameras are processed to detect activities). These events can be used again by operators or be consumed by customers. The challenge of operator placement is to calculate a placement that, for example, minimizes the end-to-end delays between the sensors and the event consumers.

To evaluate the performance of their approach, Rizou et al. [RDR11] used the *Network Emulation Testbed* to set up a network consisting of up to 200 hosts running the sensors, the event consumers and the operators. Here, the latencies between the hosts were

specified to be equal to the delays measured between the hosts in PlanetLab [CCR<sup>+</sup>03]. The benefit of the emulation system is, that these delays are exact the same for every experiment run and, therefore, allow for comparing the performance of different placement strategies.

Finally, the emulation testbed was used by Kierzkowski [Kie09] to evaluate the performance of a self-organizing and distributed file system (*SoDFS*). The basic idea is to adapt the placement of replicas to the location of readers and writers in the network to minimize the average delay for these operations. Kierzkowski used the *Network Emulation Testbed* to setup a network consisting of 31 autonomous systems generated with BRITE [MLMB01]. One virtual node running the distributed file system was assigned to each autonomous system. Using this setup, Kierzkowski was able to evaluate the *SoDFS* with different parameters in a controlled environment.

These case studies show the applicability of our network emulator for research and development of distributed applications and communication protocols. The main benefit for the users is the possibility to run unmodified applications in arbitrary user-defined networks. There was no need to modify the evaluated software to run the experiments using the network emulator. Therefore, the network emulation approach allows for minimizing the effort for preparing network experiments.

The network sizes of these experiments results from the evaluation demands and are not constraint by the emulation testbed. For example the emulation of the stream processing application using the PlanetLab topology was executed on a single physical node. With the available 16 physical nodes the network could easily be increased to 3.200 nodes (despite that PlanetLab does not contain so many nodes). Therefore, these case studies rather show the different application domains than the scalability of network emulation. A detailed performance evaluation is presented in Section 4.5 and Section 5.5.

# 4

## Efficient Node and Time Virtualization

In Chapter 2, we discussed two major building blocks for scalable network emulation: node and time virtualization. Node virtualization allows for evaluating scenarios with a number of virtual nodes beyond the number of physical nodes by partitioning the physical testbed resources. The concept of time virtualization prevents overloading the physical resources by virtually increasing the testbed resources like CPU and network.

In this chapter, we introduce concepts to increase the scalability of network emulation in terms of possible scenario sizes and resource consumption. First, in Section 4.1, we discuss the system architecture (see Section 3.2), previously published in [GMHR08, GHR10], to efficiently support node and time virtualization by minimizing the memory footprint and the communication overhead in detail. Second, in Section 4.2, we extend our network emulation tool *NETshaper* to accurately emulate network properties in a time virtualized emulation environment [GMHR08]. Third, in Section 4.3, we provide a scalable multiplexing scheme to emulate multiple virtual links using a single physical network interface. The approach is based on the diploma thesis of Schirmer [Sch11a]. Finally, in Section 4.4, we present the concept of *adaptive virtual time*, previously published in [GMHR08, GHR09a], which minimizes the experiment runtime by adapting the experiment speed to the load of the system. This chapter closes with a detailed evaluation of the presented concepts in Section 4.5 and a summary in Section 4.6.

## 4.1 Architecture for Efficient Virtual Time

As discussed in the previous chapter, scalable network emulation can be achieved by combining efficient node virtualization and time virtualization. In this section, we present our emulation architecture which allows for memory and network I/O efficient virtual nodes in the presence of transparent time virtualization. The architecture was previously published in [GMHR08, GHR10].

The remainder of this section is structured as follows: First, we introduce a hardware model of the virtual nodes and an application model of the *Software under Test* running inside these nodes. We classify the operations executed by a *Software under Test*. Second, we present our hybrid virtualization architecture supporting node and time virtualization to minimize the memory and communication overhead. Finally, we discuss approaches to efficiently connect virtual nodes to the emulation network. We show that depending on the number of CPUs and the number of network interfaces available at the testbed nodes different approaches are applicable.

### 4.1.1 Application Model and Hardware Model

The performance of distributed applications and communication protocols is heavily influenced by the environment in which the software is executed. In the case of network emulation, the properties of the environment are modeled by connected virtual nodes with emulated virtual hardware. Figure 4.1 shows the hardware model of a virtual node as well as the application model of the *Software under Test* (SuT) executed by the virtual node.

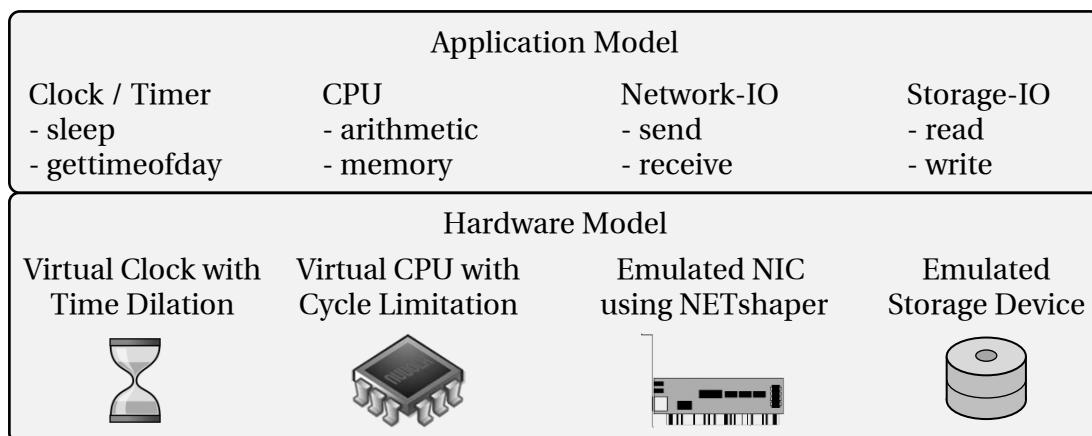


Figure 4.1: Model of the *Software under Test* and the virtual hardware

The hardware of a virtual node can be roughly divided into four components: a virtual clock, a virtual CPU, an emulated network interface and an emulated storage device. All these hardware components can be accessed by the *Software under Test*. In Figure 4.1 we have exemplarily included typical operations of a SuT to access the hardware components. In the following, we define these four types of operations in detail:

- a) *Clock* and *timer*-based operations are used to explicitly describe the behavior of the SuT over time. These operations allow for delaying other operations (e.g., *sleep*) or to access the time (e.g., *gettimeofday*). In a time-virtualized network emulator, these operations need to access a virtual clock instead of the real time clock. Running virtual nodes inside a virtual machine, allows for providing virtual time transparently to the SuT without any modifications on the SuT.
- b) The performance of *CPU*-based operations is mainly dependent on the performance of the CPU (e.g., arithmetic operations). By slowing down the rate of the virtual clocks, we are able to emulate virtual CPUs with an arbitrary speed. For any amount of resource requirement of a SuT an adequate TDF can be chosen to prevent overloading the emulation testbed. However, in time-sensitive applications, such as video players on portable devices, the available CPU resources determine the performance of the application. Here, the emulator need to provide an artificially constraint virtual CPU. Emulators like FoxyLargo [YYK08] allow for the emulation of the CPU speed. Here, the cycles of the virtual CPU that the *Software under Test* can allocate per time unit can be limited. In our application model, operations on the memory such as copying memory pages are included in this category, because their execution requires, dependent on the system architecture, significant CPU interaction.
- c) *Network*-based operations allow for communication between virtual nodes. The behavior of network I/O can be emulated by network emulation tools such as *NETshaper* [HR02]. Based on the bandwidth and delay of an emulated network device as well as the number of enqueued frames, the tool can calculate the delivery time of a frame.
- d) *Storage*-based operations refer to accessing the secondary memory (e.g., hard disc) of virtual nodes. Software, such as distributed transaction monitors, makes use of secondary memory as stable storage. Therefore, the performance of the storage-based operations can be essential for the performance of the SuT. Disk emulators [GSS<sup>+</sup>02, Gc12] allow for emulating storage devices with specific performance characteristics.

In the remainder of this thesis, we restrict our focus on *clock* and *timer*-based and *network*-based operations. The extension of our network emulator by using existing CPU and disk emulators can be the subject of future research activities.

### 4.1.2 Hybrid Virtualization Architecture

Highly scalable network emulation [GHR12] can be achieved by combining two basic concepts: node and time virtualization. However, the performance of network emulation depends on an efficient integration of these concepts. In the following, we present an architecture to efficiently combine both virtualization techniques by minimizing the memory and communication overhead.

The physical nodes of the emulation testbed are connected by an emulation network. This network is partitioned using IEEE 802.1q VLAN (*Virtual Local Area Network*) [IEE06] to form arbitrary virtual network topologies between the virtual nodes [Her05]. Our network emulation tool *NETshaper* [HR02, GMHR08] is used to connect a virtual node to the virtual network. Thereby, the emulation tool allows for emulating the network properties like bandwidth, delay and packet loss of the virtual nodes' network devices. Due to the emulated network devices on the *Data Link* layer, this approach allows for SuT located at *Network*, *Transport*, and *Application* layer. Placing the emulation tool inside the virtual protocol stack instances enables back pressure on saturation of the emulated network connection as with real network devices.

A virtual machine [BDF<sup>+</sup>03] running on each physical node<sup>4</sup> provides virtual time transparently to the operating system and the SuT inside the VM [GYM<sup>+</sup>06]. The resources of the VM are partitioned using a virtual protocol stack [KHS<sup>+</sup>03] to create virtual nodes. The processes that make up a virtual node all share the same operating system, but each of these virtual nodes has its own protocol stack instance, including sockets and routing tables. By virtualizing additional operating system components [KW00, Ope12] like process name spaces and file systems, processes of different virtual nodes are clearly separated from each other.

Due to the common operating system, all virtual nodes running inside the same VM can share resources of the operating system like caches and libraries. Therefore, this architecture minimizes the memory overhead per virtual node. At the same time, virtual protocol stacks allow for efficient communication between virtual nodes. Reference passing can be used to transmit frames between virtual nodes running inside the same

---

<sup>4</sup>Assuming a physical node with one CPU core.

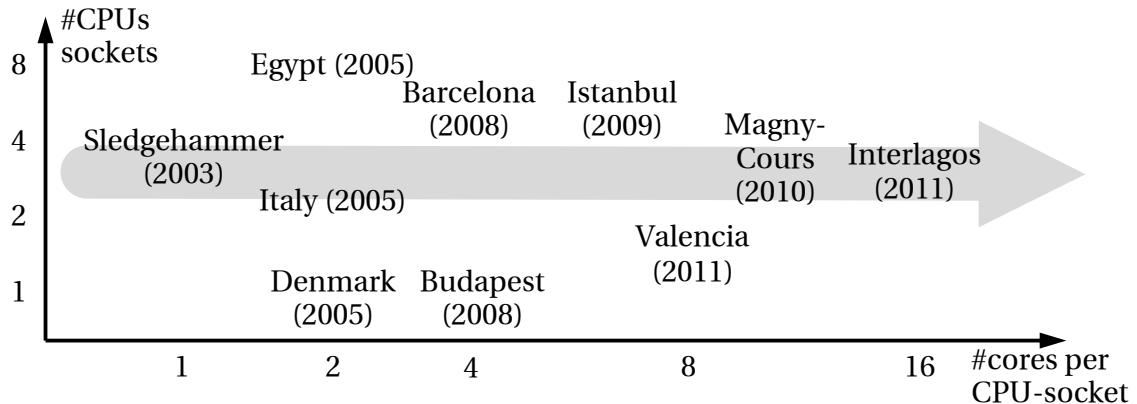


Figure 4.2: Maximum supported number of CPUs sockets and cores per CPU socket in AMD server processors<sup>5</sup>

virtual machine. Due to the common operating system, no additional, expensive context switches are required for communication.

An ongoing trend in processor design is the increasing number of CPU cores per machine to meet the performance demands of the customers. Latest CPU generations (cf. Figure 4.2) integrate up to 16 cores per CPU socket and typical machines are equipped with up to four CPU sockets. Although network emulation is inherently distributed, exploiting up to 64 cores is challenging in the presence virtual machine-based time virtualization. Generally, there are two extremes for utilizing the available resources. Either, we could assign all cores to a single virtual machine or we could run for each physical core one virtual machine with one virtual CPU. In either case, we run multiple virtual nodes inside these VMs. In the following, we abstract from CPU cores and CPU sockets and use only the term CPU. A physical node with  $n$  CPU sockets each with  $m$  cores is named a physical node with  $n \cdot m$  CPUs.

The assignment of all CPUs to a single virtual machine requires the execution of only one guest operating system per physical node and, therefore, has minimal memory overhead. Running multiple virtual nodes inside a VM in parallel, allows for utilization of multiple CPUs. However, the lower layers of the protocol stack are shared between the virtual nodes, and processing the protocol stack requires expensive locking of memory which leads to performance penalties. Since the locking effort increases with the number of CPUs, this approach would limit the scalability of the system. In addition to the scalability problem, monitoring the load of the physical node, which is required for the adaptive virtual time (cf. Section 4.4), is challenging. From the point of view of

<sup>5</sup>Advanced Micro Devices, Inc. <http://www.amd.com>

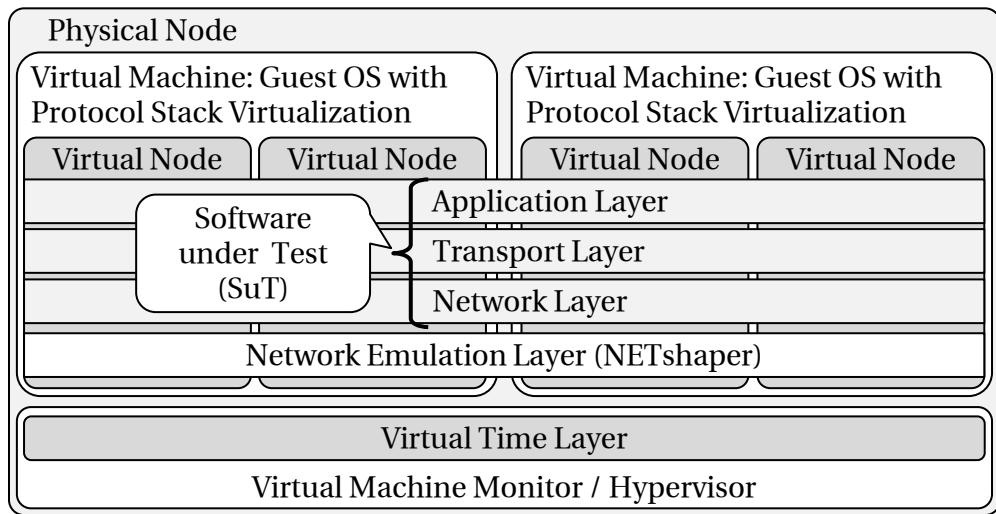


Figure 4.3: Hybrid emulation architecture: node and time virtualization

the virtual machine monitor (hypervisor), it is impossible to reliably detect overload. A situation where a single threaded application runs interchangeably on two CPUs, utilizing each by 50 %, cannot be distinguished from a situation where two applications run on two CPUs and both CPUs are 50 % idle. Although in the first situation the speed of the application is limited by the CPU (system overload), in the second situation, this is not the case. Without instrumenting the operating system inside a virtual machine, which we want to avoid to ensure realistic emulation results, it is hard to distinguish between overload and underload in such situations.

In order to be able to consider a VM as a black box and to efficiently exploit multi-core architectures, we propose to assign exactly one CPU to each virtual machine. The virtual CPU of a VM is pinned to a physical CPU, which ensures that each VM is executed on a dedicated physical CPU. Since each VM has only one CPU and runs its own operating system on exclusive memory, no memory locking between CPUs is required.

Figure 4.3 shows a physical node with two CPUs using our hybrid emulation architecture which combines virtual protocol stack and virtual machine-based virtualization. On each CPU one virtual machine is executed. The interface between the virtual machine and the hypervisor is extended by the virtual time layer. The VMs execute a guest operating system which provides virtual protocol stacks. In the figure, four virtual nodes executing the SuT are distributed to the two VMs. The network between the virtual nodes is emulated by our emulation tool *NETshaper*.

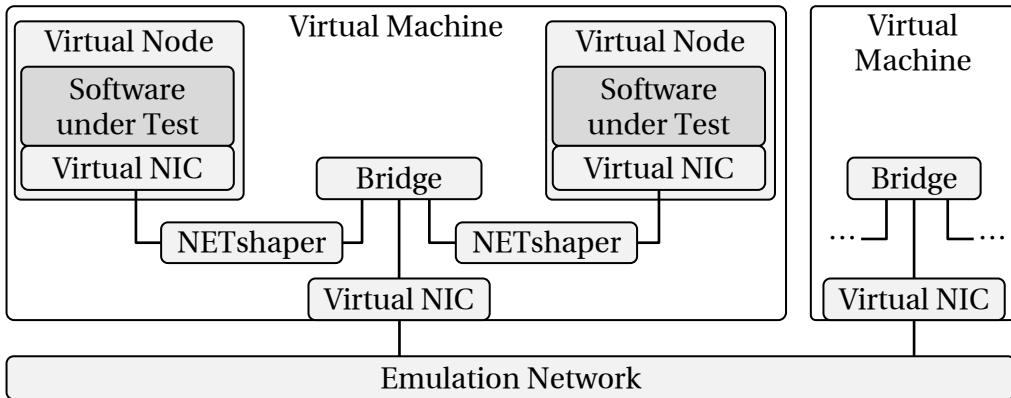


Figure 4.4: Intra virtual machine communication

#### 4.1.3 Efficient Network Access

Scalable network emulation requires efficient communication between virtual nodes. Our emulation architecture, discussed in the previous section, allows for efficient communication of virtual nodes running inside the same virtual machine. Figure 4.4 shows two virtual nodes running inside the same VM. Both nodes are attached to a software bridge to pass frames between the nodes. The software bridge is the virtual representative of the signal carrier such as a physical wire or the air in case of a wireless transmission. Our network emulation tool *NETshaper*, placed between the software bridge and the virtual network interface of a virtual node, allows for emulating the characteristics of the virtual network link.

In order to enable communication between virtual nodes running in different virtual machines, the software bridge is attached to the virtual network interface of the virtual machine. Connecting the virtual network interfaces of the virtual machines to the emulation network switch allows for communication between virtual nodes running in arbitrary virtual machines. In case of multiple virtual links that need to be connected to the emulation network, multiplexing techniques (cf. Section 4.3) are used to connect all software bridges, representing the virtual links, to the single virtual network interface of the virtual machine.

There are several techniques [YBYW08] to connect a virtual machine with the emulation network. These approaches can be divided into approaches utilizing the host to connect virtual machines with the emulation network (cf. Figure 4.5a) and approaches where the virtual machines directly access the hardware (cf. Figure 4.5b and 4.5c). Device Emulation [SVL01] is a host-based approach, where the hardware interface of a network

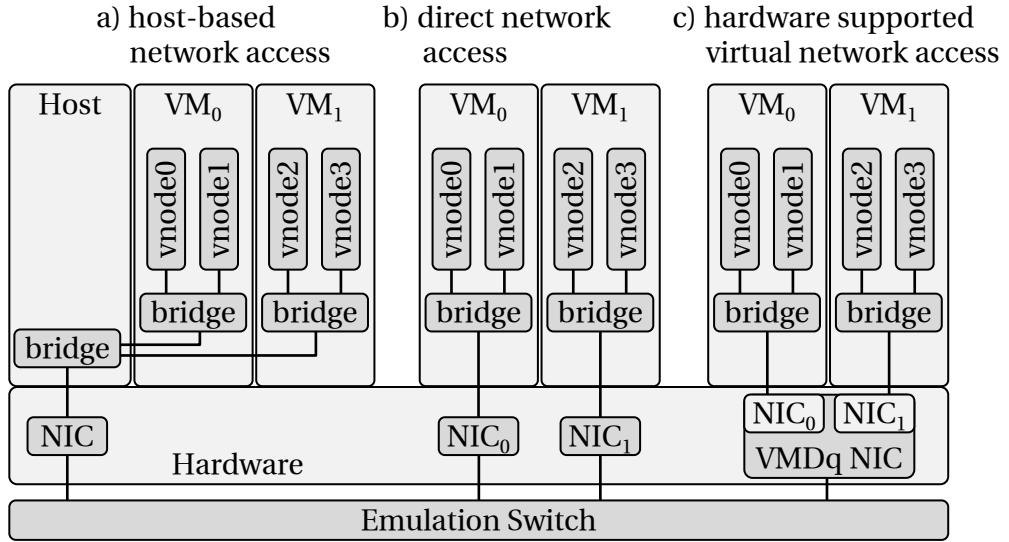


Figure 4.5: Alternatives for network access

interface card is emulated in software running inside the host operating system. An unmodified network driver running inside the VM provides the network access to the operating system inside the VM. On the host side, the emulation software can access the network interface of the host to send and receive frames. The flexibility of emulation testbed users to evaluate the performance of device drivers is paid by the large overhead of this approach [YBYW08, GWS06]. The main reason for this overhead are the required memory operations to copy frames between the memory of the host and the guest operating system.

This overhead can be reduced by paravirtualized drivers [BDF<sup>+</sup>03, Rus08], also known as virtual I/O drivers. Here, the software for device emulation and the unmodified device driver are replaced by a connected backend and frontend driver, respectively. These custom drivers are efficiently connected by shared memory. For medium speed network interface cards (1 Gbps), the network throughput of this approach is similar to the native approach, where the network is accessed from the host operating system [YBYW08, GWS06]. However, in case of high speed network interface cards (10 Gbps), the introduced CPU consumption prohibits fully utilizing the network capacity.

Almost native network performance as well as low CPU consumption can be achieved by providing direct hardware access [LUSG04, YBYW08] to the virtual machine (cf. Figure 4.5b). Since the device driver running inside the virtual machine directly accesses the hardware, each physical network interface can only be accessed by one virtual machine. As discussed in the previous section, the number of virtual machines running on

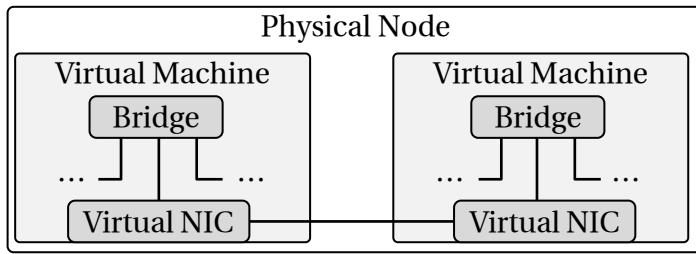


Figure 4.6: Inter virtual machine communication

a physical node is equal to the number of CPUs of a physical node. Even with network adapters with multiple network interfaces [Int12], this approach only scales for physical nodes with a low number of CPUs because of the limited number of switch ports.

Virtualization support in network devices, such as VMDq [CH07], allows for partitioning of network devices (cf. Figure 4.5c). The network device provides a set of virtual network devices which can be assigned to the VMs and directly accessed by them. Since the virtualization is done by the hardware, the performance is equal to accessing multiple physical network interfaces.

Independent of the network access method, all communication between virtual nodes running in different virtual machines requires physical network access or at least interaction with the host operating system. In case of a communication of virtual nodes running in virtual machines of the same physical node, establishing a direct link between virtual machines (cf. Figure 4.6) can circumvent the overhead associated with the physical network access. Approaches such as XenLoop [WWG08] provide a virtual network interface to each virtual machine and connect them using shared memory.

In the remainder of this thesis, we apply the *direct network access* for testbeds with only one CPU per physical node. In case of testbeds with multiple CPUs per physical node, we apply a *host-based network access*.

## 4.2 Fine-Grained Timer for NETshaper

In order to properly emulate link properties like bandwidth and delay, the network emulation tool *NETshaper* [HR02, GMHR08] is attached to the virtual network interface of a virtual node running the *Software under Test*. The basic mode of operation of the emulation tool is as follows:

For each frame to process, the emulation tool calculates the transmission time of the frame based on the bandwidth and the delay of the link as well as the transmission time of the previous frame. Thereafter, the tool starts a timer to go off at this time, and places the frame in a delay queue. The frame is removed from the queue and transmitted when the timer expires. Since our emulation tool runs inside the virtual machine, the only available timers are based on the timer interrupt with a typical granularity of 1 ms. Generally it is possible to extend the VM to support hardware timers with a higher resolution such as APIC timers [Int01]. However, this requires a large number of expensive context switches involving the hypervisor which we would like to avoid to minimize the system load.

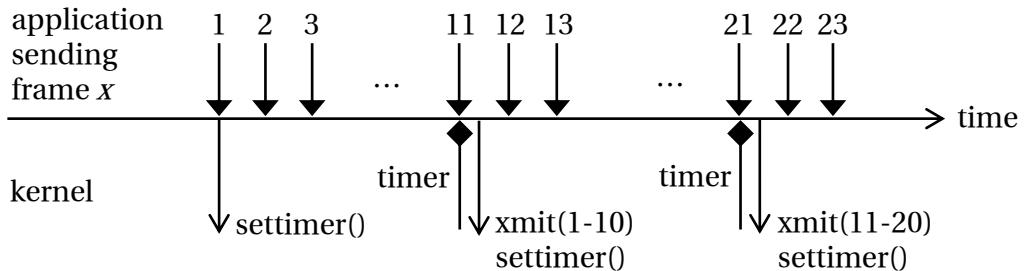


Figure 4.7: Emulation with interrupt triggered timers

For emulating high speed links ( $>1$  Gbps) the granularity of interrupt triggered timers available inside the virtual machines is too coarse. These links transmit more than a hundred frames per millisecond which results in a bursty transmission behavior. In Figure 4.7 the bursty behavior is visualized by an example of an application sending a frame every 0.1 ms at a configured propagation delay of 1 ms. Each timer activates the transmission of ten frames. Such a behavior is not only unrealistic but it also leads to load peaks which prohibits a load-aware adaptation of the virtual time (cf. Section 4.4). To avoid these effects, we use a timer triggered by events [AD99] instead of interrupts.

The basic idea behind event triggered timers [GMHR08] is to reuse existing system events, such as in network interrupts [JNVP06], to trigger the timer. We use frame level

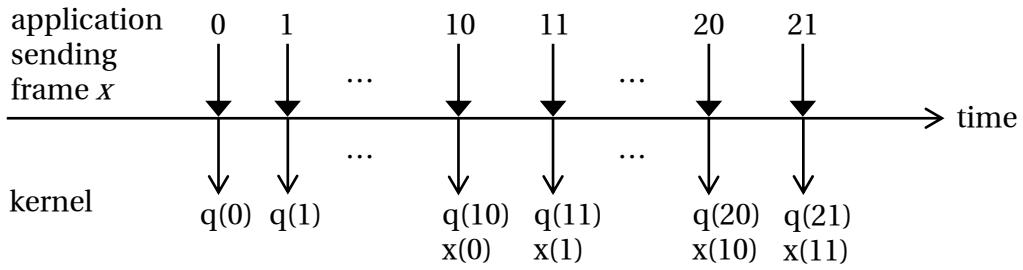


Figure 4.8: Emulation with event triggered timers ( $q$  = enqueue frame;  $x$  = transmit frame)

sending and receiving events to trigger delayed frame transmissions. Whenever a frame enters the emulation tool the delay queues are checked for frames to be transmitted.

Figure 4.8 again shows an application sending a frame every 0.1 ms. However, here an event triggered timer is used for frame delays. After the first ten frames are enqueued, transmission requests of the following frames trigger the transmission of previously delayed frames. Using this technique, in this example, one frame is transmitted every 0.1 ms.

The main benefit of this approach is that increasing the link speed increases timer granularity, too. The granularity of the timers can be further improved by sharing the events of all instances of the emulation tool in a virtual machine. In case of periods where the duration between two frame level events is higher than the configured delay, interrupt triggered timers are used as a fall back solution, which guarantees a worst case timer resolution equal to the resolution of the interrupt-based timer, which is sufficient for realistic delay emulation. Since there are only a few frames to be delayed in such situations, this approach does not result in a bursty transmission.

Additionally, the usage of event-based timers to transmit delayed frames minimizes context switches and is cache friendly (memory and processor cache), because frames are transmitted while the emulation tool is executed anyway [PBR<sup>+</sup>08]. Therefore, no other tasks of the operating system are interrupted.

## 4.3 Multiplexing of Virtual Links

Network emulation requires a configurable network infrastructure to establish virtual links between virtual nodes. In the following, we first introduce the VLAN-based emulation approach (*Virtual Local Area Network*) [Her05, Mai11] previously used in the NET project (*Network Emulation Testbed*) and discuss its limitations. Thereafter, we discuss related work in the area of virtual link emulation. Finally, we introduce our concepts of a scalable emulation scheme for the emulation of virtual links. For further details, we refer to the diploma thesis of Schirmer [Sch11a].

### 4.3.1 VLAN-based Network Emulation

In the NET project, VLAN-based network emulation [Her05] is used to emulate virtual links between virtual nodes. VLANs (IEEE 802.1q [IEE06]) are used to partition the physical network hardware. Figure 4.9 shows a simple network consisting of four virtual nodes (A, B, C, and D). Node A and B are connected by a point to point link and nodes B, C, and D are attached to the same LAN (e.g., switched network).



Figure 4.9: Example network topology

Figure 4.10 shows the emulation of this example network using VLAN-based emulation [Her05] without the use of node virtualization, i.e., one virtual node per physical node. Each virtual link (point to point link or switched network) is emulated by a unique VLAN. Frames leaving a physical node are tagged by a VLAN. The outgoing ports of the emulation switch are also tagged by a list of VLANs. The emulation switch then propagates such frames only to those ports, where the VLAN is included in the list of assigned VLANs.

The VLANs ensure that transmitted frames are only received by virtual nodes attached to the same virtual link (same VLAN). Without the use of VLANs a broadcast of node A would also be received by node C and D. The VLAN ensures that only node B receives broadcasts of node A. Since the VLAN identifier is contained in the Ethernet header, this approach does not reduce the size of MTU (*Maximum Transmission Unit*). Therefore, this approach is fully transparent to the upper layer of the protocol stack.

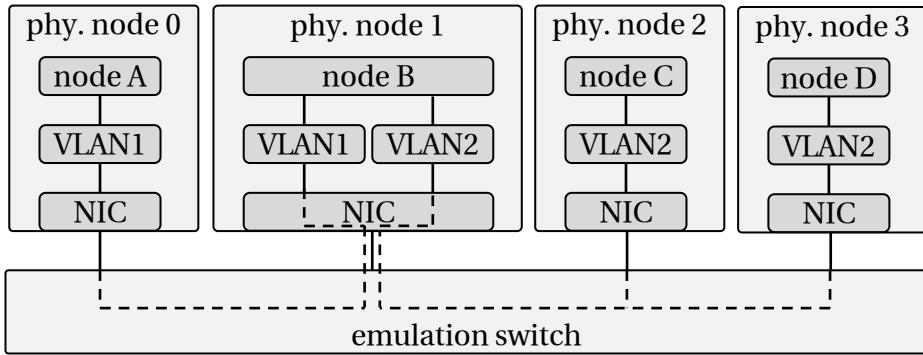


Figure 4.10: VLAN-based emulation [Her05]

Maier [Mai11] extended this approach to support node virtualization. Figure 4.11 shows the same example network emulated by two physical nodes. The virtual nodes A, C, and D are placed onto physical node 0 and virtual node B is placed onto physical node 1. Again, each virtual link is mapped to a unique VLAN to ensure that frames are only received by virtual nodes attached to the same virtual link.

The concept of node virtualization allows for increasing the number of virtual nodes and, therefore, the number of virtual links, too. Each virtual link between virtual nodes running on different physical nodes requires a unique VLAN<sup>6</sup>. However, the number of VLANs is limited and, therefore, the VLAN-based emulation limits the number of virtual links. Two factors limit the number of VLANs. First, the header field storing the VLAN identifier in a frame is limited to 12 bit ( $2^{12} = 4,096$  VLANs). Second, the amount of VLANs actually supported by hardware switches can be below this number. Table 4.1 shows the number of supported VLANs for a selection of common Ethernet switches.

<sup>6</sup>Virtual links between nodes running on the same physical node can be emulated without VLANs [Mai11].

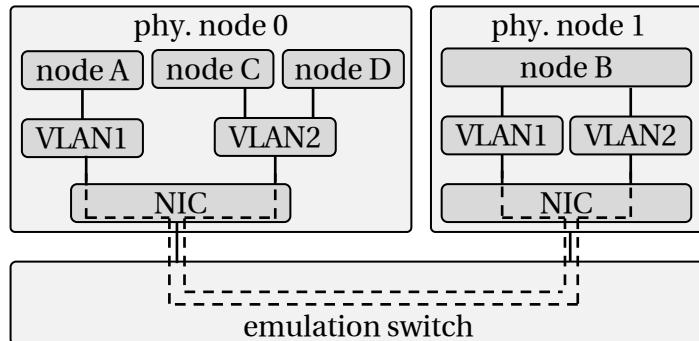


Figure 4.11: VLAN-based emulation in presence node virtualization [Mai11]

<b>Manufacturer/Model</b>	<b>#VLANs</b>	<b>#Multicast Groups</b>		<b>Reference</b>
		<b>Layer 2</b>	<b>Layer3</b>	
Arista 7148S	4,094	2,048	2,048	[Ari12]
D-link DGS-6600	4,096	1,024	1,024	[D-112]
Netgear GSM7352S-200	1,024	1,024	1,024	[Net12a]
Cisco ME 4924-10G	4,096	16,000	8,000	[Cis12]

Table 4.1: Properties of common Ethernet switches

For comparison, we have included the number of supported multicast groups on Layer 2 and Layer 3, too.

### 4.3.2 Related Work

The simplest approach to increase the number of supported virtual links makes use of additional hardware. Instead of one physical network interface per physical node connected to an emulation switch, multiple network interface cards connected to multiple switches are used. Each additional network interface (plus the corresponding switch) provides additional 4,096 VLANs. The same VLANs can be reused to emulate virtual links on different physical network interfaces. However, the additional hardware costs limit the applicability of this approach.

In V-eM [AC06] different approaches are used for the emulation of unicast and broadcast traffic. In case of unicast traffic, backward learning of MAC addresses is used. All common switches maintain a table mapping Layer 2 MAC addresses to ports. Using these table, a unicast frame is transmitted to the correct receiver. For broadcast traffic, each virtual link is mapped to a Layer 2 multicast address. On the sender, the emulation tool replaces the broadcast address by a Layer 2 multicast address. On the receiving physical nodes these frames are delivered to the virtual nodes attached to the corresponding virtual link after replacing the multicast address by the broadcast address. Common switches used for emulation have Layer 2 multicast support and, therefore, allow for filtering out physical nodes without a virtual node attached to the virtual link addressed by the multicast address. However, similar to the VLAN-based approach, the scalability depends on the number of multicast groups the switch supports. Table 4.1 shows, that this number is similar to the number of supported VLANs and, therefore, the scalability of this approach is limited, too.

MobiNet [MRBV05] and Empower [ZN03] emulate virtual links by transmitting the frames to the destination node using tunneling. In case of unicast traffic, this approach

adds overhead for additional protocol headers. These protocol headers decrease the supported MTU (*Maximum Transmission Unit*) or introduce additional overhead due to the required fragmentation. Broadcast traffic can be emulated using multicast or multiple unicasts which limits the scalability (limited switch support) or requires additional overhead, respectively.

In NEPTUNE [GBC09] virtual links are identified using unique Layer 2 addresses. In case of unicast frames, the target address on Layer 2 is used to forward the frame to the correct virtual nodes. In case of broadcast frames, the hardware switch will transmit the frame to each physical node. Here, NEPTUNE filters duplicates by enforcing non overlapping subnets on the *Network* layer.

As a conclusion, all related multiplexing schemes have either limited scalability due to the limited hardware support for multicast groups or require additional overhead due to frame fragmentation or message duplicates.

### 4.3.3 Concepts for Scalable Link Multiplexing

In this section, we present our approach to increase the number of supported virtual links. The base of our approach is the VLAN-based approach discussed on Section 4.3.1. However, instead of using one VLAN per virtual link, multiple virtual links between virtual nodes running on the same subset of physical nodes are mapped to the same VLAN. In the following, we first show how to multiplex and demultiplex the virtual links. Second, we discuss the number of virtual links supported by this approach. Finally, we compare the scalability of the extended VLAN-based approach and the previously used approach.

Figure 4.12 shows our architecture to efficiently emulate multiple virtual links using a single physical network interface. The first demultiplexer (*demux1*, see Table 4.2) is used to forward incoming frames to virtual machines hosting virtual nodes attached to the virtual link the incoming frame belongs to. The second demultiplexer (*demux2*, see Table 4.2) forwards the frames to the bridge connecting virtual nodes attached to

Source MAC address	Virtual Link	Virtual Machine
FE:14:43:D2:F1:5A	link 11, link 23 (bridge)	VM 2
C2:21:73:3D:E1:11	link 42 (bridge)	VM 1
12:34:56:78:9A:BC	link 11 (bridge)	VM 3, VM 4

Table 4.2: Table mapping MAC addresses to virtual links and virtual machines

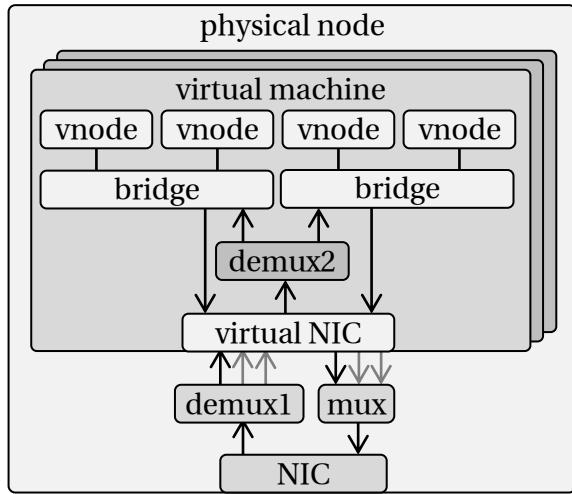


Figure 4.12: Scalable VLAN-based emulation architecture

this virtual link. In order to perform the demultiplexing, a table  $M : s \mapsto l$ , mapping the MAC address  $s$  of a virtual node to the corresponding virtual link  $l$ , is used by both demultiplexers. This allows for identifying the virtual link the frame belongs to based on the source address field of a frame. If, for example, a frame with the source MAC address C2:21:73:3D:E1:11 arrives at the NIC (see Figure 4.12) the demultiplexer *demux1* can identify the virtual machines the frames needs to be forwarded to. According to Table 4.2, it is VM 1. Inside the virtual machine the demultiplexer (*demux2*) can use the table again to identify the virtual links (link 42), emulated by bridges, and forward to frames to them. This table can be generated based on the experiment specification and can be deployed to the physical nodes while setting up the topology.

The transmission of frames involves multiplexing virtual links to a single physical network interface. Correct demultiplexing requires only the source address of a frame. Therefore, the simplest approach for the multiplexer is to transmit the frame without any additional handling. In case of unicast frames, the *MAC table*<sup>7</sup> of the Ethernet switches allows for forwarding the frame only to the switch port of the physical node hosting the virtual node. However, since the entries of the table are filled using backward learning, frames addressed to unknown addresses are flooded to all ports. The previously introduced demultiplexer can be used to discard frames received by physical nodes that do not host the virtual nodes addressed by the frame. However, the physical nodes dropping undesired frames experience additional load. Since these duplicates only occur at the beginning of the experiment, the introduced load can be neglected.

---

<sup>7</sup>The table is build up using backward learning and maps MAC addresses to switch ports.

Additionally, it is possible to build up the *MAC table* transmitting a dummy frame from each virtual node.

Broadcast frames, transmitted by virtual nodes, are forwarded by the Ethernet switch to all physical nodes. Like with unicast frames, the demultiplexers can be used to discard accidentally received frames. In case of point-to-point<sup>8</sup> links, this approach results in a large number of frames to be discarded and thus a large overhead, because each frame is received by all physical nodes, and all except of one will discard it.

The usage of VLANs avoids the accidentally received frames. One VLAN is assigned to each subset of physical nodes hosting virtual nodes of a virtual link. Multiple virtual links covering the same subset of physical nodes use the same VLAN. If the virtual nodes attached to two virtual links are executed on the same set of physical nodes  $n_1, n_2, \dots, n_i$ , then both virtual links are emulated by the same VLAN. Different sets of physical nodes, require different VLANs. The VLAN ensures that a frame is only forwarded to switch ports connected to physical nodes hosting virtual nodes of the virtual link addressed by the frame.

In case of point-to-point links, there are  $\frac{n \cdot (n-1)}{2}$  subsets of physical nodes ( $n$  is the number of physical nodes in the testbed). Therefore, the commonly supported number of 4,096 VLANs is sufficient to establish a VLAN between each pair of physical nodes for testbeds with up to 91 physical nodes. In case of collision domains covering multiple virtual nodes, such as for LAN emulation, the number of subsets is  $2^n - n - 1$ <sup>9</sup>. The number of VLANs allows for establishing a VLAN between each subset of physical nodes for testbeds with up to 12 physical nodes. However, the virtual nodes of one LAN are typically placed onto the same physical node rather than distributed over a random subset of physical nodes of the testbed and, therefore, the size of supported testbeds increases. For example, if no virtual link exists that has attached virtual nodes running on physical nodes  $n_1, n_2, \dots, n_i$  than no VLAN needs to be assigned to this subset of physical nodes.

In the following, we compare the possible scenario sizes of our link multiplexing scheme with the previously used VLAN-based approach (cf. Section 4.3.1). The previously used VLAN-based multiplexing scheme allows for emulation of up to 4,096 point to

---

<sup>8</sup>A virtual link between only two virtual nodes.

<sup>9</sup>A set of  $n$  elements has  $2^n$  subsets (including the empty and the full set). However, no VLANs are required for virtual links with virtual nodes running on a single physical node, because the switch of the testbed is not involved in the communication between the virtual nodes. Additionally, no VLAN is required for the empty set. Therefore, the  $n$  sets containing only one physical node and the empty set are removed from the  $2^n$  subsets.

point links between virtual nodes running on different physical nodes. The extended approach, allows for emulating an arbitrary<sup>10</sup> number of virtual links. For testbeds with up to 91 physical nodes, this approach introduces no additional overhead in terms of additional message transmissions. In case of testbeds beyond that limit, virtual links between 4,096 pairs of physical nodes require no additional overhead. Virtual links between additional pairs of physical nodes cause extra frame receptions.

The previously used VLAN-based emulation limits the number of supported collision domains with more than two attached virtual nodes to 4,096. The extended approach abolishes that limit for testbeds with up to 12 physical nodes without any extra overhead in terms of message transmissions. In case of larger testbeds, 4,096 subsets of physical nodes hosting collision domains can be emulated without additional overhead. Further subsets cause extra message overhead.

As a conclusion, the extended multiplexing scheme allows for increased scenario sizes while maintaining the benefits of VLAN-based emulation. The developed architecture provides an efficient link multiplexing scheme for our virtualization architecture composed of virtual machines and virtual protocol stacks.

---

<sup>10</sup>Limitations due to the memory consumption for running virtual nodes still exist.

## 4.4 Adaptive Virtual Time

Physical resources of emulation testbeds, such as the CPU capacity, the memory bandwidth, the network throughput, and the disk throughput, are limited. In presence of node virtualization, these resources are shared between multiple virtual nodes. The case where the sum of requested resources exceeds the provided resources is called *overload*. Overloading any resources at any time during the experiment may lead to biased emulation results, e.g., message transmissions experience additional, undesired delay.

Unbiased emulation results can be achieved by monitoring the resource utilization during the experiment to detect *overload* [Mai11]. In case of a detected overload, there exist two approaches to handle it: First, rerun the experiment with a different mapping of virtual nodes onto physical nodes that reduces the load on the physical nodes. Second, rerun the experiment with a reduced emulation speed using the concept of virtual time [CFH<sup>+</sup>80, GYM<sup>+</sup>06] (cf. Section 2.3). Finding a suitable node placement or emulation speed that avoids overload may require multiple experiment executions and, therefore, extends the total experiment runtime.

Additionally, varying resource requirements of virtual nodes lead to situations during the experiment run where the sum of requested resources are less than the provided resources. These situations are called *underload*. In presence of time virtualization, *underload* unnecessarily extends the runtime of experiments by leaving resources unused. Minimizing the overall experiment runtime requires to maximize utilization of available resources.

In order to maximize the resource utilization without overloading the physical resources, we provide the concept of *adaptive virtual time*, previously published in [GMHR08, GHR09a]. During the experiment, we monitor the resource utilization and adapt the emulation speed to prevent *overload* and avoid *underload*. Details on the implementation can also be found in two diploma theses by Egorenkov [Ego08] and Pakai [Pak08].

The remainder of this section is structured as follows: First, we give an overview on related work. Second, we discuss the concept of *adaptive virtual time* in detail. A detailed and expressive evaluation of our prototype implementation is provided in the subsequent section.

#### 4.4.1 Related Work

Canon et al. [CFH<sup>+</sup>80] were the first to introduce virtual time in emulations. However, no adaptive clock rate is used in their approach. Therefore, the clock rate has to be preconfigured to a value that ensures that no overload situation occurs throughout the experiment run. This leads to very conservative clock rates to avoid overload during load peaks. Besides the period of a load peak, the system does not utilize the existing resources efficiently and, thus, the experiment runs much longer than necessary.

Hybrid systems combine the benefits of network simulation and network emulation, by connecting physical nodes [Liu08b] running real implementations to a simulated network based on *Parallel Discrete Event Simulation* (PDES) [Fuj89, RFA99]. In addition, these approaches use a combination of node and time virtualization [ELL09]. However, a constant clock rate is used here too. The necessary synchronization in the simulation produces additional overhead.

Weingärtner et al. [WSHW08] have proposed an approach to conservatively synchronize multiple virtual machines to a simulation framework. Here, an experiment is evenly divided into time slices. The end of each slice constitutes a barrier to synchronize the VMs with the simulation framework. Yoginath and Perumalla [YP11] follow a similar approach. Here, multiple virtual machines running the *Software under Test* are synchronized using a discrete event simulation framework NetWrap. However, the VM-based node virtualization increases the overhead introduced by the synchronization scheme.

Emulation of arbitrary powerful virtual resources can be achieved by adapting the Linux protocol stack to use virtual time instead of real time. While Wang et al. [WK02] only use a simulation framework running on a single physical node, dONE [BVB06] uses a distributed simulation framework. In contrast to our system, dONE only supports testing of application layer implementations using the BSD socket interface [GN98].

All existing approaches either have additional synchronization overhead or only support a constant clock rate which both results in a suboptimal experiment runtime. We solve these problems by dynamically adjusting the clock rate to the current load of the system.

#### 4.4.2 TDF Adaptation Process

In this section, we introduce the concept of *adaptive virtual time*. To minimize the runtime of experiments, we dynamically adjust the virtual clock rate to the current resource demand of the experiment. The rate of the virtual nodes' clocks is slowed down by a factor named *time dilation factor*  $\tau$  (TDF) [GYM<sup>+</sup>06]. Equation 4.1 shows how the virtual nodes' clocks rate ( $R_v$ ) and the rate of a real clock ( $R_r$ ) are related by means of the TDF  $\tau$ .

$$R_v = 2^{-\frac{\tau}{10}} \cdot R_r \quad (4.1)$$

This transformation from real time to virtual time is executed whenever the virtual machine monitor updates the virtual machine's time during the periodic timer interrupt (1 kHz). Since interrupt handlers should only run for a short time to maintain responsiveness of the operating system to other events (e.g., message reception) an efficient implementation of this transformation is required. Additionally, floating point arithmetic is not accessible from the virtual machine monitor and can only be emulated by multiple integer operations. To allow an efficient implementation using integer arithmetic, we use (in contrast to the linear relation proposed by Gupta et al. [GYM<sup>+</sup>06]) a logarithmic relation between the rate of virtual time and the real time (cf. Table 4.3). Using only integer arithmetics, we can adjust the rate of the virtual clock with a step width of about 6.7 %. This granularity is sufficient for the adaptation algorithm that we will introduce in the following. In addition, without a logarithmic relation, this granularity depends on the virtual clock rate. As shown in Table 4.3, for fast rates (small value of  $\tau$ ) the granularity is coarse and it increases with slower rates (large value of  $\tau$ ).

In order to perform the adaptation of the TDF, we propose the concept of epoch-based virtual time [GMHR08]. The experiment is divided in epochs of different lengths where the TDF is constant within each epoch. Whenever the resource demand changes, an epoch switch is triggered to adapt the TDF to the load of the system.

Figure 4.13 shows the TDF adaptation scheme. Each physical node of the emulation system runs a *Load Monitor* to monitor the load of the physical node and to report it to a *central coordinator* which calculates the overall system load. The overall load is defined as the maximum over all individual node load values. This definition is chosen to ensure that no physical node is overloaded at any time.

Using the overall load, the coordinator determines a new TDF and initiates an epoch

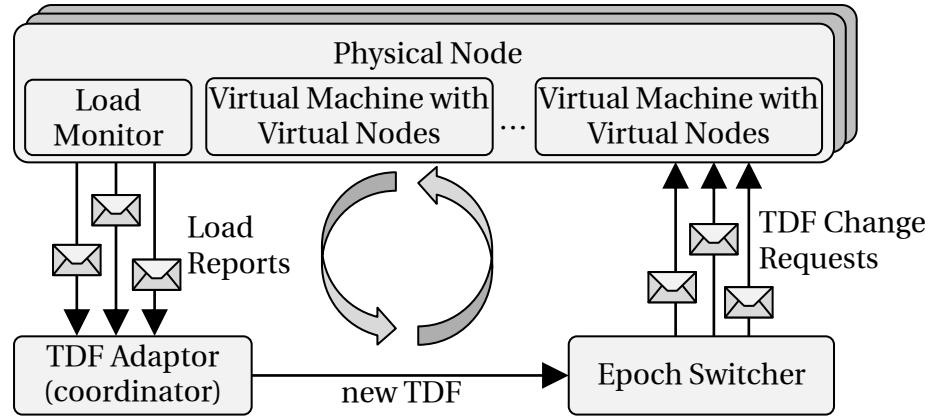


Figure 4.13: Feedback loop for TDF adaptation

switch. The *Epoch Switcher* is used to distribute the new TDF to the physical nodes and to perform an epoch switch. In the following, each component (*Load Monitor*, *TDF Adaptor* and *Epoch Switcher*) is discussed in detail.

### Distributed Load Monitoring

The *Load Monitor* is used to measure the load of a physical node and report the load to the *TDF Adaptor*. Overloading any resources of the system can bias emulation results. In network emulators the most limiting resource is the CPU. Accessing I/O resources like network or disc are typically very CPU intensive. Mechanisms such as TCP offloading [FHL<sup>+</sup>05] and RDMA (*Remote Direct Memory Access*) [HGLP07, XOP11] allow for a direct

$\tau_n \rightarrow \tau_{n+1}$	linear [GYM <sup>+</sup> 06]			logarithmic $\left(2^{\frac{-\tau}{10}}\right)$ [GMHR08]		
	$\frac{R_{r,n}}{R_{v,n}}$	$\frac{R_{r,n+1}}{R_{v,n+1}}$	slowdown factor	$\frac{R_{r,n}}{R_{v,n}}$	$\frac{R_{r,n+1}}{R_{v,n+1}}$	slowdown factor
1 → 2	1.000	0.500	50.0 %	0.933	0.871	6.7 %
10 → 11	0.100	0.091	9.1 %	0.500	0.467	6.7 %
20 → 21	0.050	0.048	4.8 %	0.250	0.233	6.7 %
30 → 31	0.033	0.032	3.2 %	0.125	0.117	6.7 %
40 → 41	0.025	0.024	2.4 %	0.063	0.058	6.7 %
10 → 20	0.100	0.050	50.0 %	0.500	0.250	50.0 %
20 → 30	0.050	0.033	33.3 %	0.250	0.125	50.0 %
30 → 40	0.033	0.025	25.0 %	0.125	0.063	50.0 %
40 → 50	0.025	0.020	20.0 %	0.063	0.031	50.0 %

Table 4.3: Granularity comparison for linear and logarithmic time dilation factors

communication between the main memory and the I/O devices which reduces the load of the CPU. However, in the presence of virtual machines the I/O access is still CPU intensive [BDF<sup>+</sup>03]. Our experiments verified this assumption and showed that the CPU capacity limits the network bandwidth between virtual nodes. The throughput of a transmission between two virtual nodes running on different physical nodes is constraint by the CPUs and not the physical network.

Therefore, load definition that only focuses on the CPU also avoids overloading the I/O in our system. However, the approach can be easily extended to consider also network or disc resources. First, we measure the resource utilization for each resource ( $load_{CPU}$ ,  $load_{network}$ ,  $load_{disc}$ ) and combine them using the following equation:

$$load_{combined} = \max(load_{CPU}, load_{network}, load_{disc}) \quad (4.2)$$

For the following description of our epoch-based virtual time, we assume the CPU-based metric.

As discussed in Section 4.1.2, for each physical CPU one virtual machine is executed. Each CPU is only accessed by one assigned virtual machine and the host operating system. The host operating system is used to establish communication between virtual machines. The load of the host operating system can be distributed arbitrarily to the CPUs  $1, \dots, i$ . We used a load monitor in the *Virtual Machine Monitor* (VMM) to measure the percentage of CPU time consumed by the virtual machines ( $L_1^{VM}, \dots, L_i^{VM}$ ) and host operating system  $L^{host-os}$ . Here, 100 % corresponds to the capacity of one CPU. The load of a physical node is defined as the load of the CPU with the maximum load. Preventing overload of this CPU, prevents overload of the other CPUs, too. Therefore, we define the load of a physical node  $load_{CPU}$  as follows:

$$load_{CPU} = \max \left[ \max_{1..i} (L_i^{VM}), \frac{1}{i} \cdot \sum_{1..i} (L_i^{VM} + L^{host-os}) \right] \quad (4.3)$$

The term  $\max_{1..i} (L_i^{VM})$  determines the resource requirements of the maximum loaded virtual machine. Since each virtual machine runs on an exclusive CPU, this term acts a lower bound of the maximum loaded CPU. In addition to the virtual machine, one host operating system is executed on each physical node. The host operating system can access all CPUs and, therefore, is executed on any CPU with free resources. The second term calculates the portion of the overall load (execution of the host operating system and the  $i$  virtual machines) that is at least assigned to each CPU (assuming a

uniform distribution of the load to all CPUs). This value constitutes a second lower bound. The maximum of both terms determines the load of the maximum loaded CPU and, therefore, defines  $load_{CPU}$ .

Virtual machine monitors provide a per virtual machine statistic, which counts the number of used CPU cycles  $c(t)$ , which is typically used for scheduling the virtual machines. Requesting this value at two points in time ( $c(t_1)$  and  $c(t_2)$ ) allows for calculation of the load  $L^{VM} = \frac{c(t_2) - c(t_1)}{t_2 - t_1} * \frac{1}{C^{CPU}}$  ( $C^{CPU}$  defines the number of cycles a CPU can execute per time unit). Similarly the load of the host operating system  $L^{host-os}$  can be measured. To meter the time between the measurements with a sub-microsecond granularity, we use the time stamp counter register of the processor (TSC), which is increased on every CPU clock cycle.

The length of the sampling interval has a large effect on the performance of load monitoring. Short intervals are required for a fast reaction to load changes, but also result in a large number of load reports. Transmission and processing of large amounts of load reports would overload the coordinator and, therefore, limit the scalability. To limit the amount of load reports, we use three mechanisms: *adaptive sampling*, *threshold-based discretization* and *hysteresis-based state changes*. These mechanisms effectively and substantially reduce communication overhead for reporting.

*Adaptive sampling* adjusts the length of the sampling interval (time period between two consecutive load reports) to the currently used TDF. For a higher TDF (slower virtual time) a longer sampling interval is chosen. The idea behind this is that overload situations develop proportionally slower when the virtual time runs slower. Therefore, the sampling interval may be increased without taking the risk of missing any relevant change. To illustrate this effect, we consider an emulation example consisting of one instance of the SuT with a timer that fires every 10 ms. To accurately emulate the SuT the system must have free resources in every 10 ms interval of virtual time to execute the timer. In case the system runs at real time, the load of the system needs to be monitored at least at 10 ms intervals to detect overload. In case the system runs 10 times slower than real time, the monitoring interval can be increased by factor 10. With this increased interval, a possible overload during any 10 ms interval of virtual time can still be detected. Therefore, we increase the sampling interval linearly with the TDF.

*Threshold-based discretization* maps the possible load values of a physical node to the four states *load panic*, *load warning*, *reasonable load*, and *underload* (cf. Figure 4.14) using three thresholds ( $\Theta_P$ ,  $\Theta_W$ , and  $\Theta_U$ ). The load monitor determines the state locally and only in case of a state change, a load report is sent to the *TDF Adaptor*. *Underload*

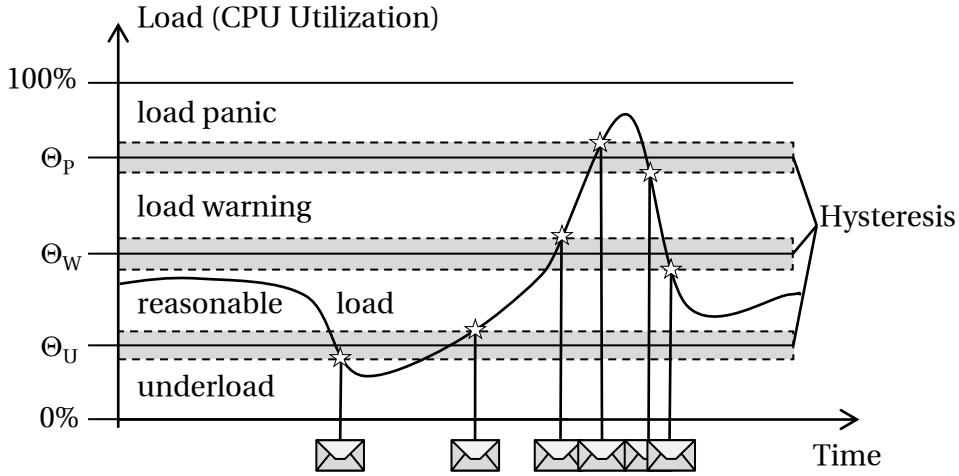


Figure 4.14: Thresholds used for TDF adaptation

indicates that there are unused resources and, therefore, virtual time could be accelerated. Analogously, the two states *load panic* and *load warning* signal that resource consumption is becoming too high. When the system is in one of these states, virtual time has to be slowed down. The two thresholds  $\Theta_P$  and  $\Theta_W$  are used to differentiate between light and heavy *load*. We describe the different reaction on these states in the next section and determine the values for the thresholds  $\Theta_P$ ,  $\Theta_W$ , and  $\Theta_U$  during the evaluation in Section 4.5.2.

*Hysteresis-based state changes* are used to avoid oscillation between two states which causes a high number of load reports. A state change is only triggered if load exceeds the threshold and its surrounding hysteresis range (cf. Figure 4.14).

### TDF Adaptation

Based on the load reports, the time dilation factor (TDF)  $\tau$  may need to be adjusted. The *TDF Adaptor* achieves this adjustment by means of a very simple proportional feedback control mechanism that is shown in Algorithm 1. Whenever the system load is outside the *reasonable* range, the algorithm adapts the TDF to reach the *reasonable load* state. As long as the system load is in state *load warning* or *underload*, a small adjustment  $S_s$  is applied (added or subtracted) to avoid overshooting the *reasonable load* state. If there is a fast increase in load, this adjustment will not suffice and the system will eventually reach the *load panic* state. In this situation, a larger step size  $S_l$  is used for the adjustment in order to decrease the load quickly and avoid overload. If this results in an *underload* situation, the algorithm will gradually decrease the TDF again to speed

up virtual time.

After each adjustment, the algorithm needs to wait for feedback from the load monitor to see whether the load is back in the state *reasonable load*. Due to the adaptive sampling rate of the load monitor, the time until the feedback arrives depends on the current TDF. Therefore, we dynamically adjust the waiting time ( $T_s$ ) to the used sampling interval.

In case of temporarily constant resource demands, the utilization can be kept steady at any level between the  $\Theta_U$  and  $\Theta_W$  thresholds in the state *reasonable load*. For good resource usage, however, the system utilization should be near the  $\Theta_W$  threshold. Therefore, we decrease the TDF in the *reasonable load* state, too. However, the speed of this adjustment is very low, through a waiting time  $T_l$  of an order of magnitude larger than the waiting time  $T_s$ . In combination with the hysteresis around the thresholds, the additional adjustments caused by the oscillation around  $\Theta_W$  introduce an insignificant overhead.

Our evaluation shows that the introduced algorithm prevents overload and underload for scenarios with and without changes of resource requirements.

---

**Algorithm 1:** TDF adaptation process

---

**input:**  $state, \tau_n$

```

1 while true do
2   if  $state \neq reasonable\_load$  then
3     if  $state = load\_panic$  then
4        $\tau_{n+1} = \tau_n + S_l$ 
5     else if  $state = load\_warning$  then
6        $\tau_{n+1} = \tau_n + S_s$ 
7     else if  $state = underload$  then
8        $\tau_{n+1} = \tau_n - S_s$ 
9     end
10    sleep  $T_s$ 
11  else if  $state = reasonable\_load$  then
12     $\tau_{n+1} = \tau_n - S_s$ 
13    sleep  $T_l$ 
14  end
15 end

```

---

## Epoch Switching

After determining the TDF for the next epoch, a mechanism is required for propagating the new value to the physical nodes. To ensure a fast reaction to upcoming overload, the time between detecting the resource demand and the actual TDF change must be as small as possible. Since the time to compute the new TDF is negligibly small, we need to minimize the time for transmitting load reports and TDF change requests. In addition, unbiased emulation requires all virtual clocks to run at the same rate at any time. Therefore, we need mechanisms to minimize the difference in propagation times of TDF change requests. A third problem related to epoch switching is the occurrence of message loss which cannot be detected in time.

We have developed a protocol [Ego08] for minimizing the propagation time of TDF change requests and load reports. The basic assumption behind this protocol is that all nodes are connected to a LAN. We are using the previously mentioned control network of the cluster. The delay of TDF change requests and load reports using this network consists of several components: network transmission delay, packet processing time in the protocol stack, and delay in queues. The time to transmit a frame in the network is insignificant because it is below  $200\ \mu\text{s}$  and has small variability. The processing time in the protocol stack is a magnitude below the transmission time and can be ignored as well. Most of the message delay is caused by waiting in egress and ingress queues of the physical nodes and the switch. In order to limit these delays, we are using priority queues based on *Type of Service* (TOS) of IP QoS<sup>11</sup> and prioritize TDF change requests and load reports. A last source of delay are the hardware based FIFO queues inside the network interface cards (NICs). Since we cannot change these queues, we are limiting the traffic on these interfaces to 95 % of the link capacity to keep the queues empty. Using these mechanisms, the maximum packet transmission delay can be reduced below 2 ms and message loss can be prevented with a very high probability.

Variations of the arrival and processing time of the TDF change requests may result in virtual clocks running slightly out of synchronization. Especially in long running experiments these multiple small delays can sum up to a significant error. To avoid this and keep the virtual clocks synchronized, the coordinator simulates its own virtual clock by applying the TDF change requests, too. Each TDF change request contains the virtual time of the coordinator. In case of a deviation from the virtual time of the coordinator, the execution of the next TDF change requests can be slightly delayed. Using this approach, the virtual clocks of all virtual nodes are synchronized. Our evaluations (see

---

<sup>11</sup>Ethernet switches can evaluate the payload of Layer 2 frames and read the TOS header field of IP.

Section 4.5.2) show a maximum clock skew of less than 5 ms and a clock skew of less than 2 ms for 99.5 % of time.

## 4.5 Evaluation

In the following, we first evaluate the accuracy of our emulator to emulate link properties such as bandwidth and delay. Second, we investigate the overhead of our emulation architecture. Third, we subject the concept of adaptive virtual time to a detailed performance evaluation.

### 4.5.1 Architecture for Efficient Virtual Time

We evaluate our proposed emulation architecture as follows. First, we present the software and hardware base of our experiments. Second, we evaluate our prototype with respect to the accuracy of bandwidth and delay emulation. Finally, we determine the introduced overhead of the emulation architecture. Therefore, the communication and memory overhead of the prototype is measured.

Xen [BDF<sup>+</sup>03] in version 3.1.0 using Linux 2.6.18<sup>12</sup> acts as foundation of our prototype implementation. Including OpenVZ [Ope12] into Xen provides virtual nodes based on virtual protocol stacks [KHS<sup>+</sup>03] and virtualization of name spaces and file systems [Sch00, KW00]. The operating system of the virtual machine is based on Debian<sup>13</sup>. In order to achieve maximum performance, all background services of the Debian system are disabled. Essentially, only the Linux kernel, *init*<sup>14</sup> and *sshd*<sup>15</sup> are running. The virtual nodes run a minimized custom Linux distribution. In order to minimize the memory footprint of a virtual node, all background services are disabled, too. After a start of a virtual node, only the default *init* process is running. We extended the virtual machine interface of Xen to provide guests a virtual time instead of a real time. Finally, we extended the *Virtual Ethernet Device Driver (veth)* of OpenVZ by the functionality of *NETshaper* [HR02] to emulate network properties such as bandwidth, delay, and message loss.

All evaluation benchmarks are performed on a cluster consisting of 64 PC-nodes. Each PC is equipped with an Intel Pentium 4 2.4 GHz processor, a Gigabit Ethernet adapter

---

<sup>12</sup>The Linux Kernel Archives <http://www.kernel.org>

<sup>13</sup>Debian GNU/Linux <http://www.debian.org>

<sup>14</sup>System process and parent of all processes of a Linux-based system.

<sup>15</sup>SSH (secure shell) daemon used to configure the experiment.

and 512 MB of main memory (RAM). A second 100 Mbps network interface is used to setup and control the experiments (*control network*) without influencing the experiment traffic on the Gigabit network interface (*emulation network*). In the following evaluations, we used VLANs to multiplex multiple virtual links to the single emulation network interface (cf. Section 4.3.1). We divide the memory into 64 MB for dom0<sup>16</sup> and 429 MB for domU<sup>17</sup>. The memory assigned to dom0 could be further decreased by running a custom minimal Linux system, similar to the operating system running inside the domU. However, this constant amount does not affect the evaluation results.

In the following, we evaluate the accuracy of the emulation layer. First, we measure if the emulation layer is able to enforce a configured bandwidth faithfully. Therefore, we set up a scenario with two connected virtual nodes in two variations. One variant uses a single physical node hosting both virtual nodes and the other uses two physical nodes, each with one virtual node. We configure the bandwidth of the link with different values ranging from 64 kbps to 100 Gbps without any additional delay. To measure the maximum throughput of the emulated link, we use the *netperf*<sup>18</sup> tool in UDP mode. It generates load according to configured send and receive buffers of 64 kB and an Ethernet MTU of 1,500 bytes.

As shown in Figure 4.15, the measured throughput corresponds, in both scenario variants, to the configured bandwidth. Note that, due to the used hardware, for high speed links (100 times faster than the used network hardware) an emulation running at real time is not possible. Therefore, we increased, the TDF to avoid overloading the physical nodes.

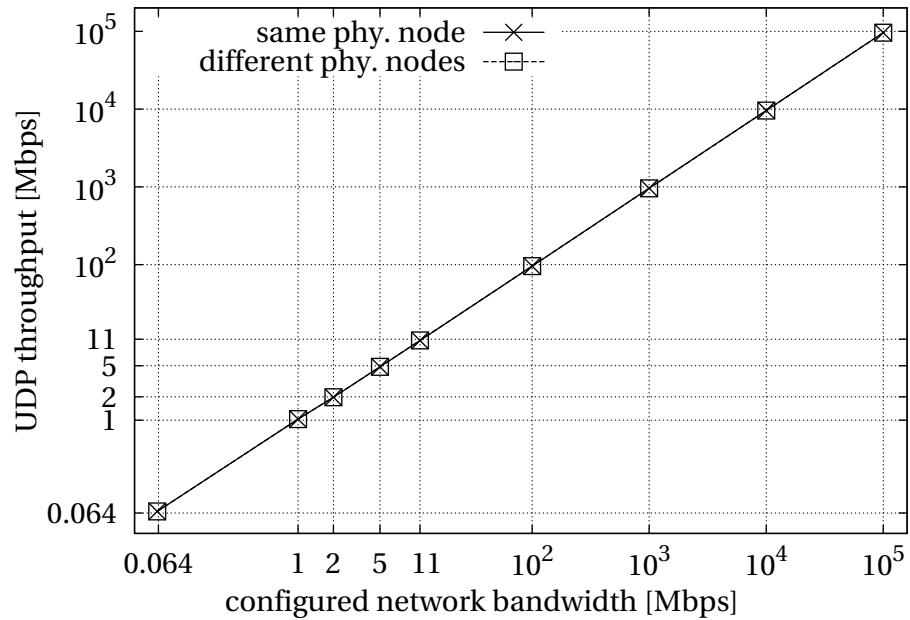
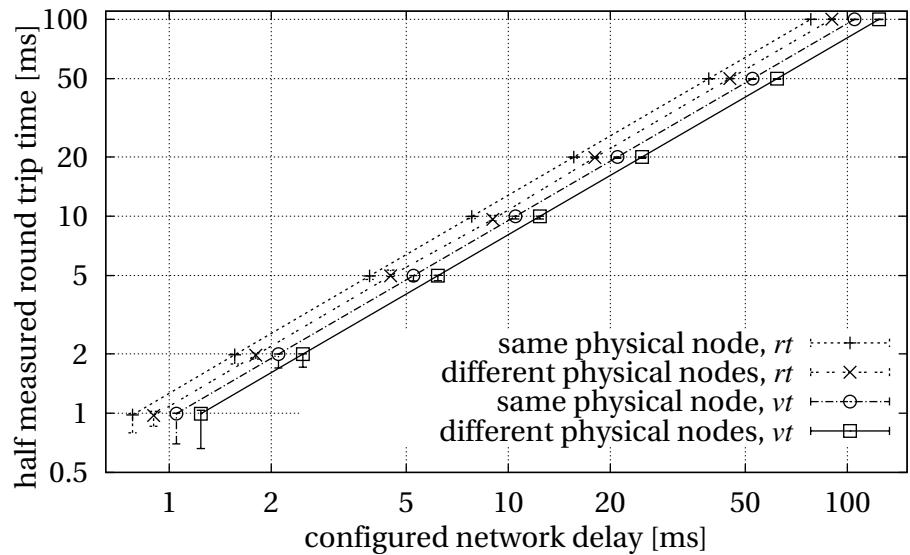
Next, we examine if the emulation tool faithfully reproduces configured delays. Again, the scenario consists of two connected virtual nodes. The virtual link between the virtual nodes has a configured bandwidth of 100 Mbps and a variable delay between 1 ms and 100 ms. We use the *ping* tool to generate ICMP ECHO requests<sup>19</sup> to measure the RTT (*Round Trip Time*) between the virtual nodes. Variations of this scenario use one or two physical nodes to host virtual nodes on same or on different physical nodes, respectively. Additionally, we run the experiment with two different TDF values *rt* (real time) and *vt* (virtual clock rate is  $\frac{1}{10}$ th of real clock rate). Figure 4.16 shows the average measured RTT and the interval of the minimal and maximal value for

<sup>16</sup>dom0 is a privileged domain in Xen. The dom0 is typically used to run device drivers and to manage (e.g., start and stop) the virtual machines.

<sup>17</sup>Name for a virtual machine in the Xen terminology

<sup>18</sup>*netperf* <http://www.netperf.org>

<sup>19</sup>Internet Control Message Protocol (ICMP) RFC 792, 1981

Figure 4.15: Accuracy of bandwidth emulation (*netperf* in UDP mode)Figure 4.16: Accuracy of delay emulation (ping)<sup>20</sup>

100 measurements per configuration<sup>20</sup>. As can be seen in the figure, the average values and the maximum values of the measurement are almost equal to the configured delay of the links. The minimal measured value, denoted by the error bars in Figure 4.16,

<sup>20</sup>All four variants have same configured delays. To increase readability, they are plotted side by side.

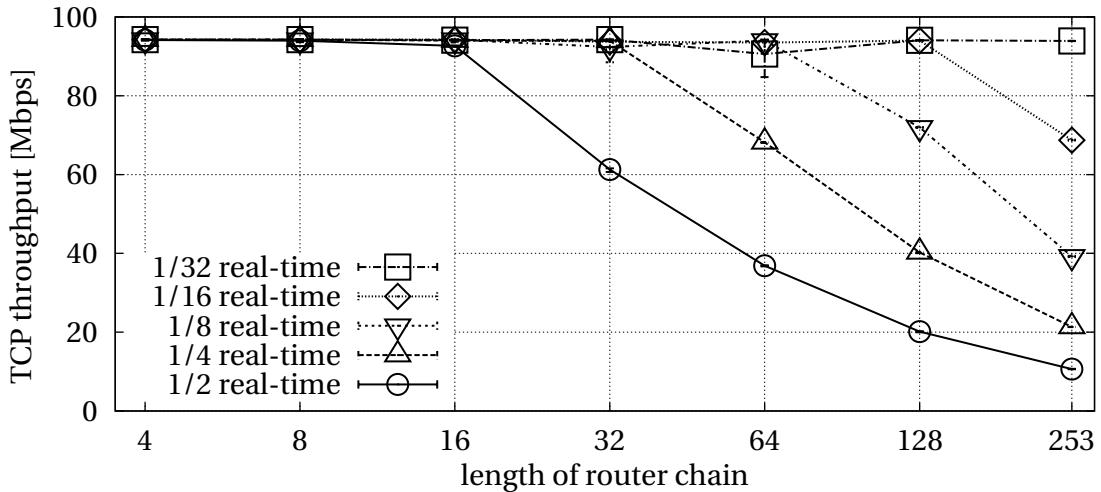


Figure 4.17: Experiment execution speed vs. scenario size

deviates slightly ( $\sim 0.25$  ms for a configured delay of 1 ms<sup>21</sup>) from the configured delay. For each measurement, only one packet is transmitted in both directions of the link and, therefore, our fine-grained timers cannot be used to increase the accuracy of the timer. Nevertheless, with a maximum deviation of about 0.25 ms, we can conclude that our emulation tool is able to accurately emulate the delays of the links.

In the following, we determine the overhead of our emulation architecture in terms of communication effort and memory consumption. To measure the communication effort, we use our network emulator to measure a TCP flow through a chain of routers in an emulated network. The scenario consists of two physical nodes, one hosting a router chain and the other a sender and receiver of a TCP connection. The sender is connected to one end of the router chain and the receiver to the other end. All links in the scenario are configured with a bandwidth of 100 Mbps without any additional delay. We run the experiment with different chain lengths ranging from 4 to 253 routers. The maximum TTL (*Time to Live*) of IP packets prohibits longer chains. We measure the maximum TCP throughput for different rates of the virtual clocks ranging from  $\frac{1}{2}$  of real time to  $\frac{1}{32}$  of real time. We use *netperf*<sup>22</sup> with configured send and receive buffers of 64 kB to transmit data using *sendfile*<sup>23</sup> for 10 s.

The results of this experiment are visualized in Figure 4.17. With all evaluated execution speeds up to 16 routers can be emulated without overloading the physical resources.

<sup>21</sup>Due to the logarithmic scale of the y-axis, the deviation seems to be larger for smaller emulated delays.

<sup>22</sup>*netperf* <http://www.netperf.org>

<sup>23</sup>Linux system call to transfer data between file descriptors (<http://kernel.org/doc/man-pages/online/pages/man2/sendfile.2.html>)

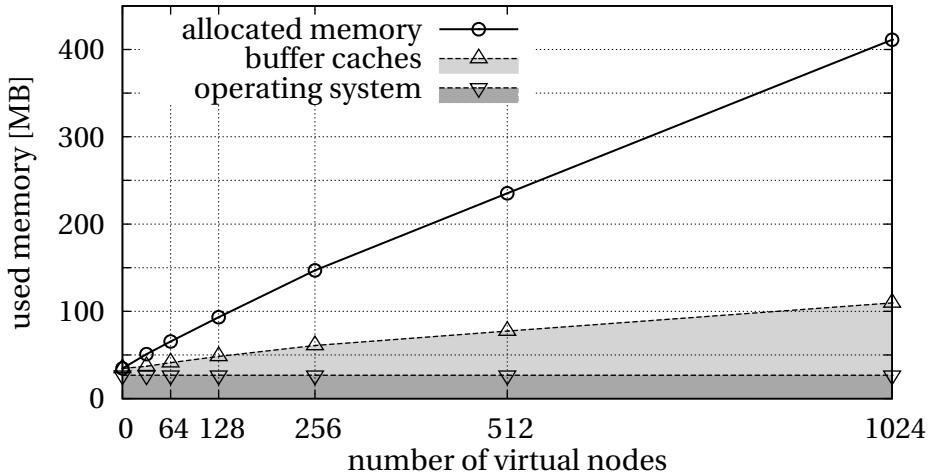


Figure 4.18: Memory consumption of virtual nodes

An overload of the system results in a drop of the TCP throughput. The system is then unable to process the frames in time, which results in frame loss and TCP will throttle the data rate. As can be seen in the figure, slowing down the executing speed of the experiment, allows for a faithful emulation of a longer router chain. Particularly, slowing down the executing speed by a factor of 2 doubles the router chain length without a data rate reduction. As a conclusion, by slowing down the execution speed of the experiment, we can emulate scenarios with arbitrary communication demand.

Besides the communication demand, the scalability of an emulation system heavily depends on the memory overhead. To evaluate the memory overhead of our approach, we create a scenario with an increasing number of virtual nodes attached to a single network. Figure 4.18 shows the required memory usage for this experiment. Here, we distinguish between the total allocated memory, the memory for caches, and the memory allocated by the operating system.

The operating system of the virtual machine including the Linux kernel requires about 27 MB and each virtual node increases the memory consumption by about 300 kB. Since we are interested in the memory overhead, no *Software under Test* is executed by the virtual nodes. In comparison, a virtual node based on Xen requires a minimum of 6 MB of memory [BDF<sup>+</sup>03]. As shown in Figure 4.18, the memory footprint increases linearly with the number of virtual nodes. This allows us to run over a thousand virtual nodes on a single physical node which is equipped with half a gigabyte of main memory. Please note that the main memory is shared by hypervisor (VMM), dom0 (64 MB), and the virtual machine.

### 4.5.2 Adaptive Virtual Time

In order to evaluate the performance of the adaptive virtual time, we integrated the concepts for load monitoring, TDF adaptation and epoch switching into our prototype. The prototype, running on the NET Cluster equipped with 64 nodes (Intel Pentium 4 2.4 GHz, 512 MB RAM, 1 Gigabit NICs), is based on Xen [BDF<sup>+</sup>03] version 3.1.0 running Linux Kernel 2.6.18 inside a virtual machine (domU in Xen jargon) and inside the control domain (dom0). In addition, *OpenVZ* [Ope12] is used to create virtual nodes inside the virtual machine. The mechanisms for adaptive virtual time were implemented as *Linux Kernel Modules* [SBP07] (LKMs) running in dom0 to minimize latencies for epoch switching.

The evaluation is structured as follows: First, we briefly discuss the chosen parameters for the load monitoring and the TDF adaptation. Then, we investigate the achieved resource utilization. Finally, we exemplify how to evaluate the performance of a routing daemon in a large scenario using our network emulator.

An extensive search of the parameter space using scenarios with different resource requirements has been performed to identify a configuration which generally minimizes experiment runtime and ensures unbiased results for our network emulator. The determined thresholds of the load monitor are:  $\Theta_U = 50$ ,  $\Theta_W = 70$ , and  $\Theta_P = 90$ . The adaptive sampling interval ranges from 5 ms for a TDF of 0 to 200 ms for a TDF of 100. TDF adjustments with a step width  $S_s$  of 1 and  $S_l$  of 20 give best results for the TDF adaptation. This configuration is used for the following experiments.

To quantify the achieved level of resource consumption, we are emulating a chain of routers routing two TCP flows (cf. Figure 4.19). The experiment is executed on two physical nodes. On the first one, two virtual nodes are running the TCP sender and receiver of the first flow  $f$  (*foreground flow*). This flow is routed through the chain of routers with different lengths. The routers are hosted by the second physical node.

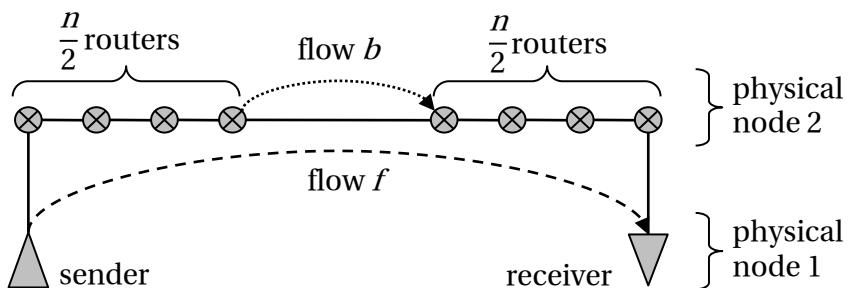


Figure 4.19: Evaluation scenario for TDF adaptation

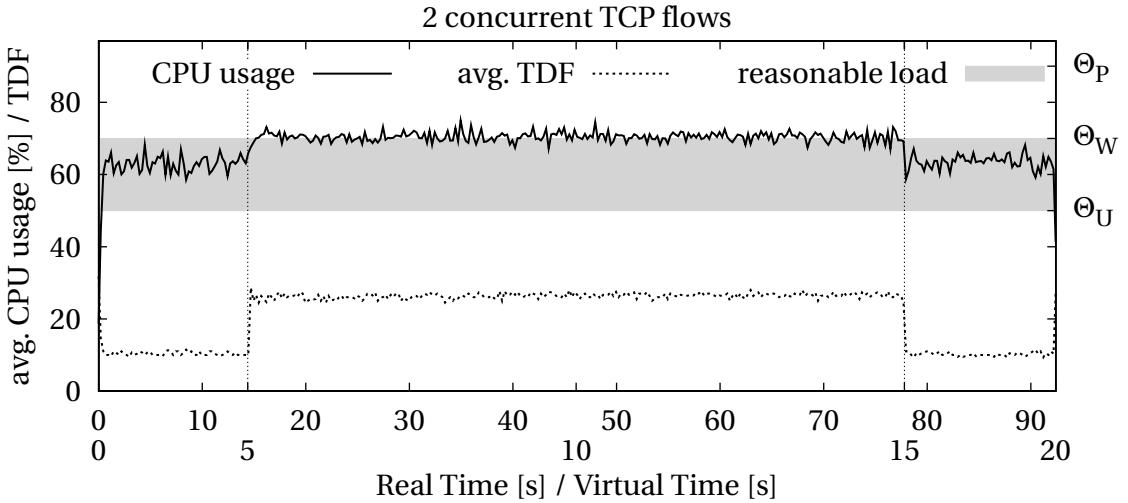


Figure 4.20: Load-based TDF adaptation

Additionally, one link of the router chain is used by a second flow  $b$  (*background flow*). The emulated network between the virtual nodes has a bandwidth of 1 Gbps except for the first and last link which have 100 Mbps. During each experiment, we run the TCP flow  $b$  for 20 s of virtual time. 5 s after the experiment start, we run the flow  $f$  for 10 s and measure the achieved throughput. In addition, the resource usage on both physical nodes is measured during the experiment. For each router chain length, the experiment is repeated 50 times.

Figure 4.20 shows the CPU utilization of the physical node running a chain of 32 routers. The time axis has two scales: the upper scale is the real time and the lower scale is the virtual time. Running only the flow  $b$  (first and last 5 s of virtual time of the experiment) requires the system to run with a TDF of about 10 ( $2^{-\frac{10}{10}} \cong \frac{1}{2} \cdot \text{real time}$ , cf. Section 4.4.2) to keep the CPU utilization inside the *reasonable load* range denoted by the gray area. Running flow  $f$  between 5 s and 15 s of virtual time increases the resource requirements. In order to prevent overload, the system automatically adapts the TDF to a value of about 27 ( $2^{-\frac{27}{10}} \cong \frac{1}{6} \cdot \text{real time}$ , cf. Section 4.4.2). As flow  $f$  stops, the system adapts the TDF back to the original value.

Figure 4.21 shows the measured results for different lengths of the routers chain. The evaluation metrics are: the achieved TCP throughput, the load of the physical node running the router chain, and the average TDF. Although these measurements have different scales, we show them in a single graph to increase clearness of display. For comparison, we have also included the results for the experiment without flow  $f$ . The TCP throughput allows for rating the accuracy of the emulation by comparing the

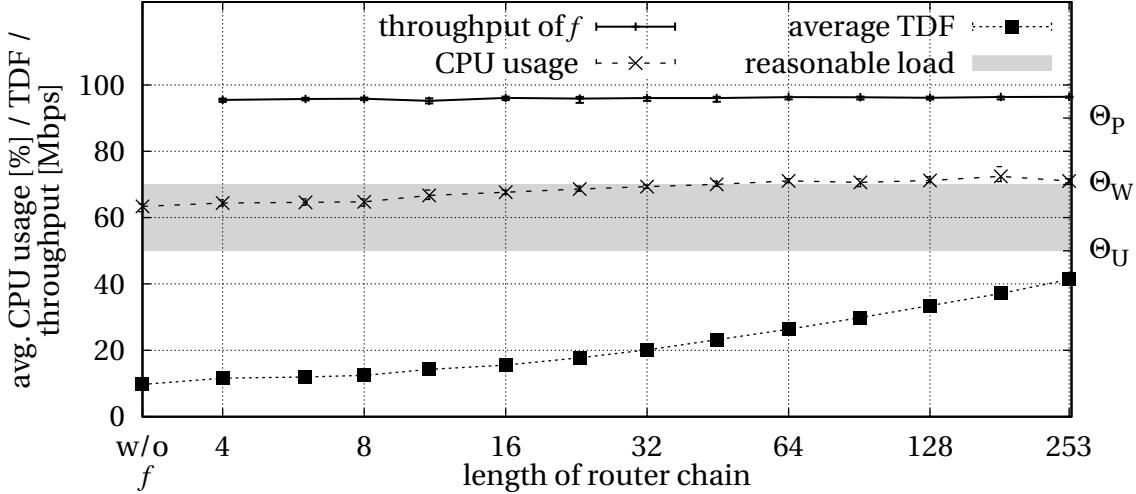


Figure 4.21: Effectiveness of the TDF adaptation

measurements with the TCP throughput in real environments. In the emulation as well as in measurements in real environments, TCP is able to achieve about 96 Mbps throughput and, therefore, we can conclude that the emulation results are not biased. The experiments are repeated 50 times. The deviation from the average is about +0.78 % and –1.34 % in the worst experiment run and, therefore, negligible.

As shown in the figure, for up to eight routers the resource utilization mainly results from flow  $b$  and, therefore, the load as well as the TDF are constant. As the number of routers increases, the resource requirements for flow  $f$  increase likewise. Since each router basically does the same, the load increases linearly with the number of routers. At a length of about 11 routers, the flow  $f$  consumes a significant amount of the CPU and, therefore, the system needs to slow down the virtual time, resulting in an increase of the TDF.

The gray area in Figure 4.21 marks the *reasonable load* range. For the experiment to exhibit minimal runtime, the resource utilization should be near the upper bound of the *reasonable load* range. As the figure shows, the load of the physical node hosting the routers approaches this limit and stays below the threshold as desired. For shorter router chains, the low TDF results in a small sampling interval (cf. Section 4.4.2) which makes the system more sensitive to short load peaks. These load variations can cause false positives of *overload warning* messages and, finally, a temporary suboptimal TDF. However, the sensitivity is required to prevent overload situations in the presence of short load peaks.

In the next experiment, we investigate the scalability aspects with respect to the used

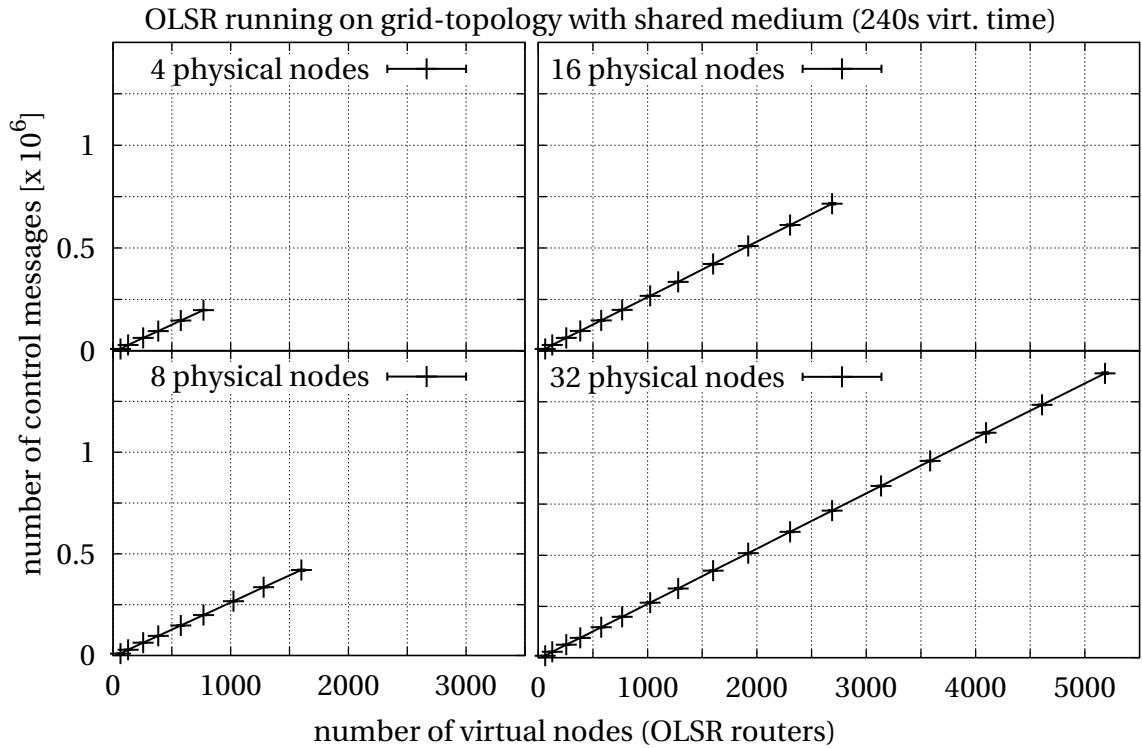


Figure 4.22: Number of control messages sent by virtual routers running OSLR

number of physical nodes used for the emulation. In this experiment, we emulate a MANET (*Mobile Ad-Hoc Network*) with an increasing number of virtual nodes. Each virtual node runs the OSLR protocol (*Optimized Link State Routing*) using the unmodified version of *olsrd*<sup>24</sup>. The nodes are arranged in a grid topology. Due to the configured transmission range, each virtual node can only directly communicate with its four neighbors. To verify that emulation results are not biased, we compare the number of control messages of the routing protocol transmitted in an experiment. This number is depicted in Figure 4.22 for four different setups with 4, 8, 16, and 32 physical nodes and an increasing number of virtual nodes running uniformly distributed on them. As the number of virtual nodes increases, the amount of control messages should increase linearly since each node emits such messages to its neighbors periodically.

If results were biased by overload situations in the experiments, this would mean that the *olsrd* instances would not be able to emit the required control messages in time. Messages would be delayed or dropped. As a result, the linear increase would change and a kink would appear at the point where the number of virtual nodes gets too high. Figure 4.22 clearly shows no such sign of an overload situation throughout the entire

<sup>24</sup>An Adhoc Wireless Mesh Routing Daemon. <http://www.olsr.org>

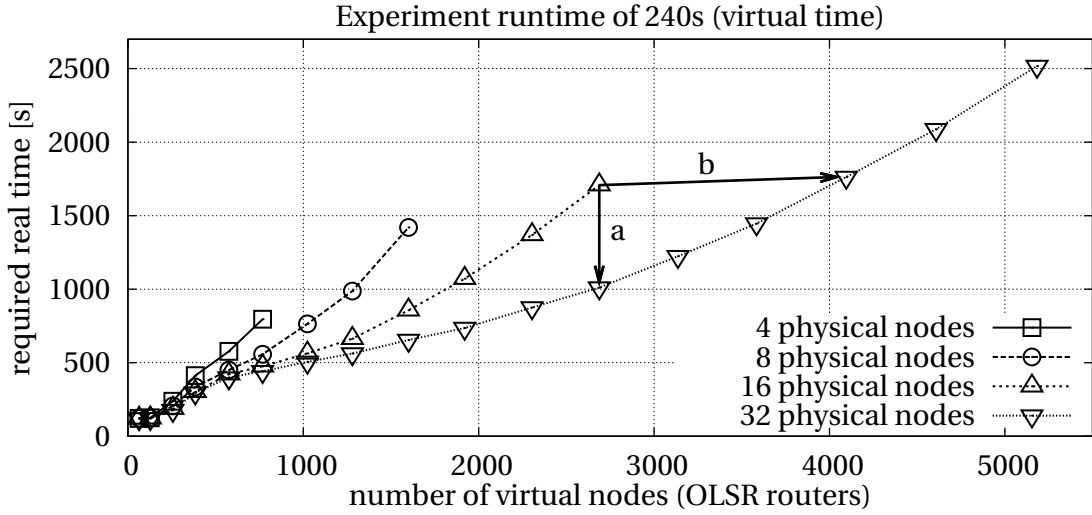


Figure 4.23: Required real time to emulate scenario with virtual routers running OLSR

range despite the increase in virtual as well as physical nodes.

Figure 4.23 shows the time required for running the OLSR scenario with different numbers of physical nodes. For low numbers of virtual nodes (< 300), increasing the number of physical nodes does not produce a notable effect as these experiments can easily be run using 4 and 8 physical nodes. However, the figure shows that, for a larger number of virtual nodes, the resources of all physical nodes are efficiently exploited. For example, with 2,700 virtual nodes, doubling the resources from 16 to 32 physical nodes can reduce the required experiment time by about 40 % if the number of virtual nodes is kept constant (cf. Arrow a in Figure 4.23). Conversely, through doubling the resources, the number of virtual nodes and, thus, the size of the scenario can be increased by 50 % without increasing the experiment time (cf. Arrow b in Figure 4.23).

Finally, we are investigating the number of load reports sent by the physical nodes to the coordinator to adapt the TDF to the load of the system and the resulting number of TDF changes. As can be seen in Figure 4.24 and Figure 4.25, for small numbers of virtual nodes (< 300) only a small number of load reports are required, because the system is most time in the *underload* state. As the number of virtual nodes increases, the resource requirements of the scenario increase, too. As a reaction to the increasing resource requirements, the system needs to adapt the TDF to avoid overload. In order to minimize the experiment runtime, the system aspires at a load near the border between *overload* and *reasonable load* state. The change between these two states requires load reports. However, as can be seen in the figure, our mechanisms to reduce the number of load reports (cf. Section 4.4.2) are effective and cause about 200 load reports per

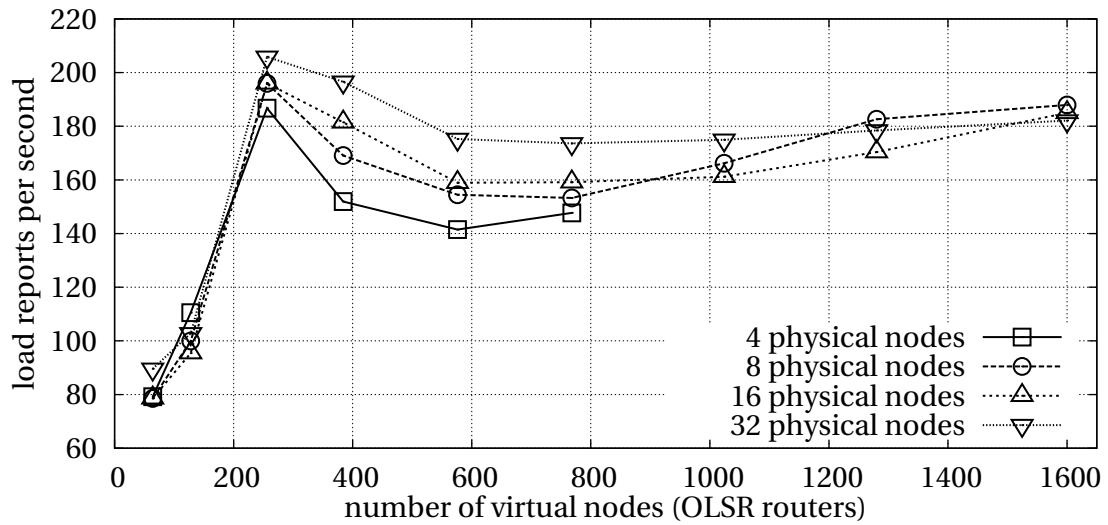


Figure 4.24: Number of load reports

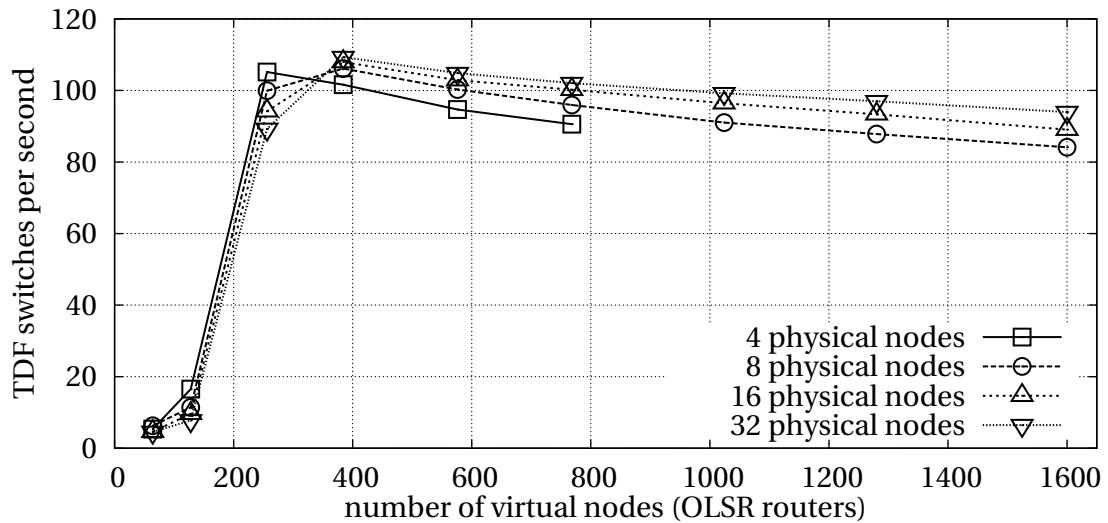


Figure 4.25: Number of TDF switches

second for all physical nodes. The number of TDF switches is at about 100 switches per second. The figures also show that the number of load reports and TDF switches are almost independent of the number of physical nodes used for the experiment. Due to the low number of load reports and TDF switches, the TDF adaptation results in almost no measurable load on the coordinator. Therefore, the coordinator can easily perform the TDF adaptation for multiple experiments in parallel.

Finally, we investigated the synchronization of the virtual nodes' clocks. Since the clock drift results from small variations of the arrival and processing time of the TDF change

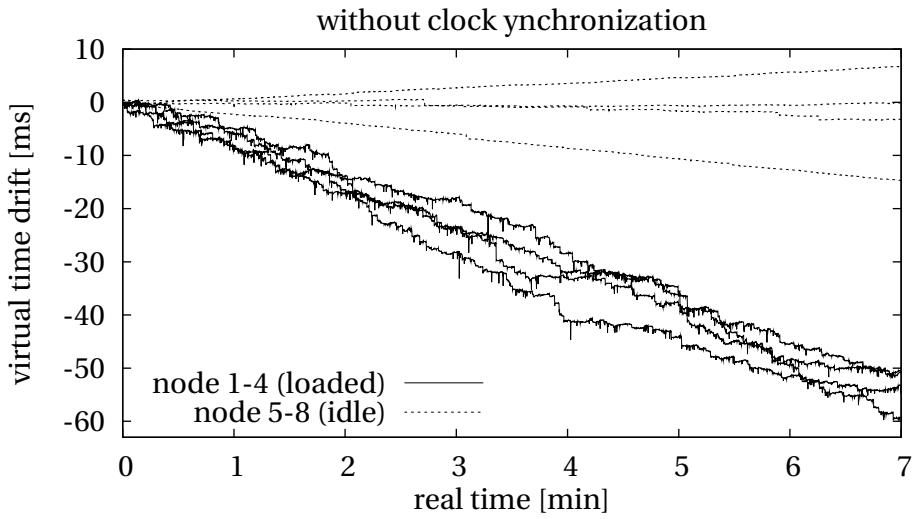


Figure 4.26: Clock drift without synchronization protocol

requests, these variations increase with the load of the virtual machine. Therefore, we evaluated the clock skew in a scenario where four physical nodes have no load (running a SuT that sleeps forever) and four physical nodes are heavily load (running the aforementioned OLSR scenario). Each physical node runs one virtual machine. In case of the loaded physical nodes, 200 virtual nodes are executed inside each virtual machine. In the following we name the physical nodes without load *idle* and the physical nodes with load *loaded*.

Figure 4.26 shows the drift of the virtual time for the 4 idle and the 4 loaded physical

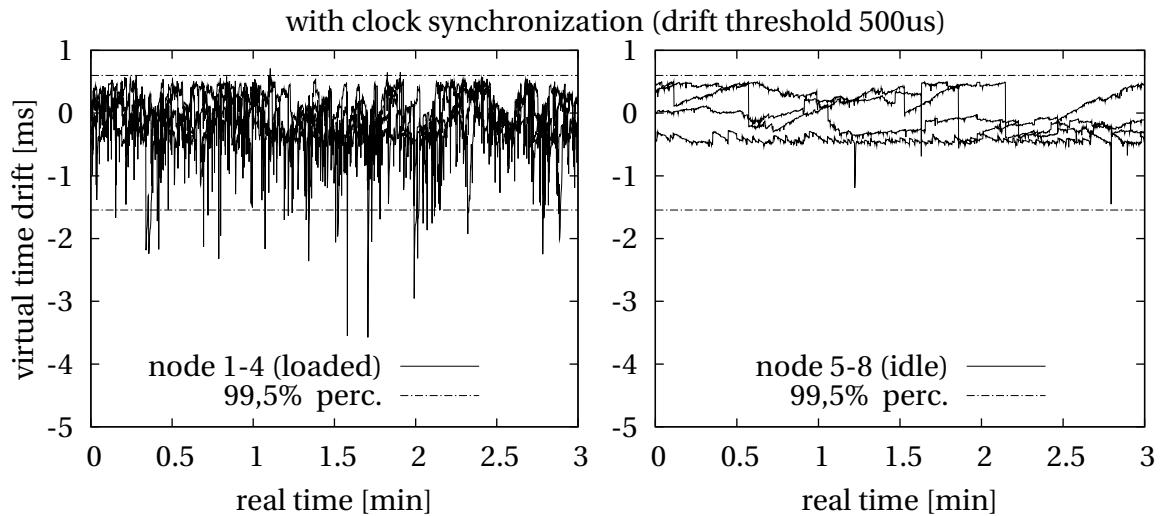


Figure 4.27: Clock drift with synchronization protocol

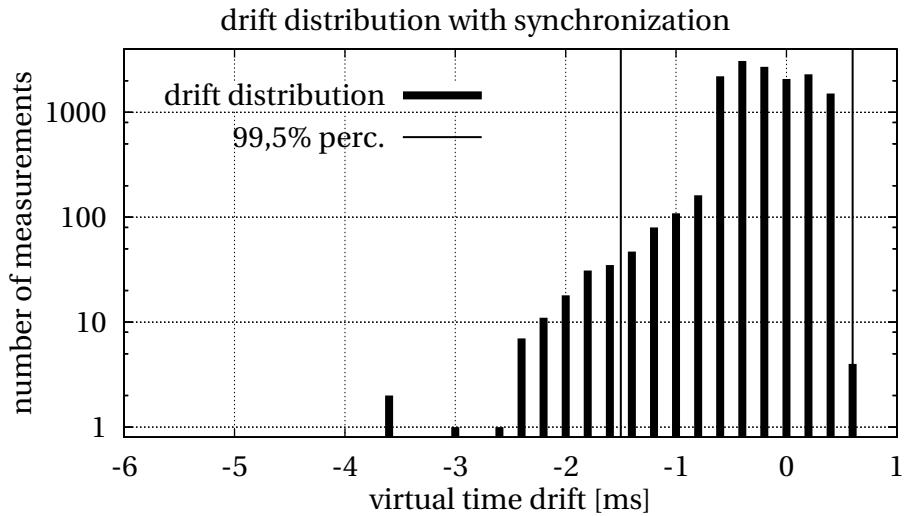


Figure 4.28: Distribution of clock drift

nodes without the synchronization protocol. As can be seen in the Figure, the drift increases during the experiment execution for all nodes. However, in case of loaded nodes the drift increases about 5 times faster than the drift of the idle nodes. Figure 4.27 shows the drift in the same scenario with the clock synchronization protocol (see Section 4.4). To increase readability we show the loaded and idle nodes separately. The Figure shows, that the drift varies faster for the loaded nodes. However, the synchronization protocol can effectively limit the drift for the load and idle nodes.

In order to quantify the synchronization accuracy, we included the 99.5 % percentiles (dashed lines in the Figure 4.27) and showed the distribution of the clock drift in Figure 4.28. As can be seen in the figures, the introduced synchronization protocol allows for limiting the drift to an 5 ms interval. Moreover, the drift remains within a 2 ms interval for 99.5 % of time. In conclusion, with the synchronization protocol the drift does not increase over time and, therefore, the protocol allows for synchronized clocks in long running experiments, too.

## 4.6 Summary

In this chapter, we introduced our emulation architecture to efficiently provide node virtualization and time virtualization in the presence of multi-core CPUs [GMHR08, GHR10]. In order to minimize the communication overhead and the memory overhead, we applied the concept of virtual protocol stacks for node virtualization. We argued that virtual protocol stacks extended by name space partitioning and file system virtualization allow for virtual nodes that are isolated from each other.

In order to cope with the resource requirements of multiple instances of the *Software under Test* on each physical node, we applied the concept of time virtualization. Here, we can reduce the resource requirements by slowing down the execution speed of the experiment. To minimize the runtime of an experiment we have developed the concept of *adaptive virtual time* [GHR09a]. Here, the load of physical nodes is monitored and overload or underload of physical nodes is reported to a coordinator. Based on these load reports, the coordinator accelerates or slows down the execution speed of the experiment.

During the evaluation of the presented concepts, we first showed the accuracy of our network emulator. Here, we showed that the system is able to accurately emulate links from slow dial-up links to future high speed connections. We also showed that the system can faithfully emulate the delay of links. Second, we investigated the scalability of our system by measuring the memory and communication overhead. The evaluation shows that our system has an overhead per virtual node of about 300 kB, which allows for running up to a thousand virtual nodes on a physical node equipped with 512 MB memory. Finally, we evaluated the *adaptive virtual time* concept. Here, we showed that our adaptation algorithm is able to counterbalance varying resource requirements of a network experiment and can achieve a high resource utilization. Additionally, we showed that our system scales with the number of physical nodes. Additional physical resource can be used to reduce the runtime of an experiment or to increase the emulated virtual network.



# 5

## Experiment Configuration

The placement of the virtual nodes onto the physical nodes determines the performance of network emulation. From the perspective of testbed users as well as testbed operators, experiments with minimal runtime are desired. In order to minimize the experiment runtime, we have developed the workflow depicted in Figure 5.1. In a first step, *NETplace* (previously published in [GHR10, GHR12]) calculates an initial placement that minimizes the experiment runtime. The input of *NETplace* consists of the testbed specification (e.g. number of physical nodes, CPUs per physical node, and CPU capacity), the network topology, and the expected average resource requirements of the *Software under Test* (SuT). As a second step, we setup the network topology in the network emulator and deploy the SuT. Finally, we execute the SuT on the virtual nodes and evaluate the properties of the SuT.

In this basic workflow, resource requirements of the SuT deviating from the average may

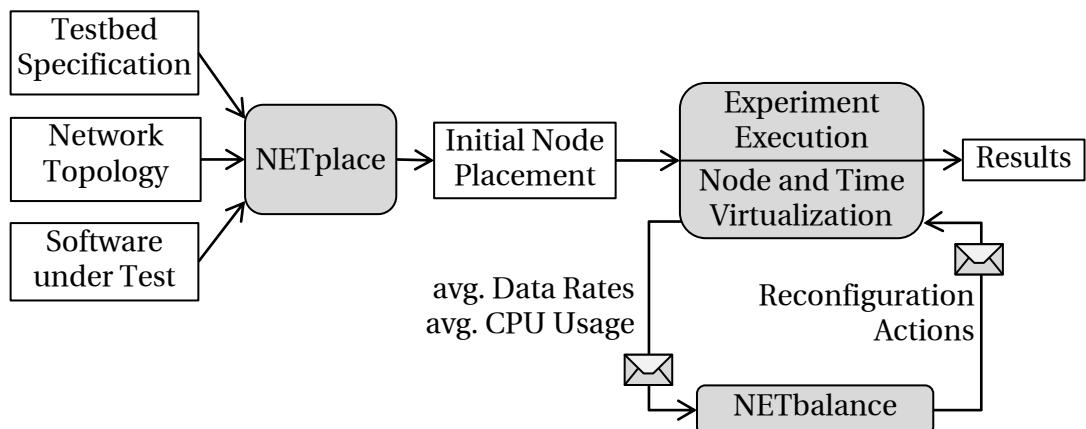


Figure 5.1: Experiment workflow using initial placement and dynamic reconfiguration

lead to temporary suboptimal placements which can lead to an extended experiment runtime. *NETbalance*, previously published in [GHR11, GHR12], extends this workflow to minimize the experiment runtime in scenarios with varying resource requirements by adapting the placement of virtual nodes. At runtime, *NETbalance* detects changes in the resource requirements of virtual nodes. These changes trigger the recalculation of the virtual nodes' placement. We transform the current placement into the optimized placement by the migration of virtual nodes between virtual machines. For extended details on the implementation of *NETbalance*, we refer to the diploma thesis of Bartmann [Bar11].

The remainder of this chapter is structured as follows. First, we introduce in Section 5.1 the configuration interface of the *Network Emulation Testbed* to specify network experiments. Second, we present our testbed model that allows for calculating the runtime of an experiment based on the placement of the virtual nodes in Section 5.2. This model acts as a basis of our approaches *NETplace* and *NETbalance* to minimize the experiment runtime, which are discussed in Section 5.3 and Section 5.4, respectively. A detailed evaluation of the presented concepts is given in Section 5.5. The chapter is concluded by a summary in Section 5.6.

## 5.1 Experiment Specification

Before running an experiment in our *Network Emulation Testbed* (NET), the experiment needs to be specified. In NET this task is supported by two independent approaches. First, there is an object-oriented description language [Gra11] based on the scripting language Ruby<sup>25</sup>. Second, we have developed a graphical user interface (GUI) using the *Eclipse Rich Client Platform*<sup>26</sup>. In the following, we provide a brief outline of both approaches.

### 5.1.1 Textual Experiment Specification

The base of our textual experiment specification is an object-oriented framework [Gra11] reflecting the components of the network experiment. Active network elements, such as routers, access points, mobile devices, and hosts, that can execute the *Software under Test*, are abstracted by the class *VNode* (virtual node). These virtual nodes can be connected by passive network elements, such as an Ethernet switch or radio in case of wireless communication. The passive network elements are model by the class *CollisionDomain*. The properties of these connections, such as bandwidth and delay, can be specified using the properties of the class *NIC* (network interface card).

In order to efficiently specify network experiments, our framework allows for an automatic configuration of the network layer. Therefore, the framework assigns IP addresses to each virtual node and calculates routing tables to provide communication between each pair of nodes. Besides the automatic configuration, it is also possible to configure the network layer manually.

In case of the wireless communication, the automatic configuration allows for determining the properties of the radio links between virtual nodes based on the location of a virtual node. The properties are calculated using pregenerated radio propagation maps [Ste08]. An extension of the framework developed by Schuh [Sch11b] allows for mobility emulation. Here, the framework adapts the properties of the wireless links according to the position of the mobile nodes during the experiment run.

Figure 5.2 shows an example of a network topology. The topology consists of four virtual nodes: a host, a router, an access point, and a phone. The host and the router are connected by an Ethernet switch. An access point is connected to the router via a direct

<sup>25</sup>Ruby scripting language: <http://www.ruby-lang.org>

<sup>26</sup><http://www.eclipse.org/home/categories/rpc.php>

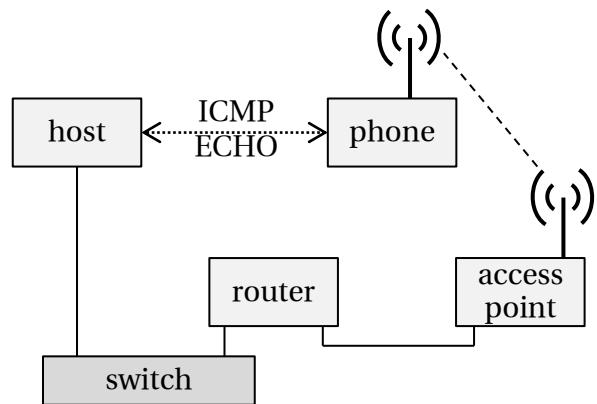


Figure 5.2: Sample network topology

cable. A phone is connected to the access point via wireless communication. In this simple experiment we initiate an ICMP ECHO request<sup>27</sup> to measure the delay between the phone and the host. Algorithm 2 shows a script file using our framework to setup and run this experiment.

<sup>27</sup>Internet Control Message Protocol (ICMP) RFC 792, 1981

**Algorithm 2:** Sample script to setup a topology with a phone connected to an access point, which is connected to a LAN by a router.

---

```
1 #!/usr/bin/ruby
2   /* load the testbed configuration */ 
3 require 'tvee'
4 NET.cleanup
5   /* setup a switched LAN with 1 host and 1 router */ 
6 switch=CollisionDomain.new
7   /* setup a host connected by 100Mbps and 1ms delay */ 
8   host=VNode.start(NET.tvee[0])
9   switch.attach_node(host.nic[0])
10  host.nic[0].configure(100000, 0, 1, 0)
11  /* setup a host connected by 1Gbps and 1ms delay */ 
12  router=VNode.start(NET.tvee[0], nil, 2)
13  switch.attach_node(router.nic[0])
14  router.nic[0].configure(1000000, 0, 1, 0)
15  /* Setup an access point with one phone */ 
16  air=CollisionDomain.new
17  /* setup the phone at position 1010/1030 */ 
18  phone=VNode.start(NET.tvee[0])
19  air.attach_node(phone.nic[0])
20  phone.nic[0].enable_manet_mode
21  phone.set_position(1010, 1030)
22  /* setup the access point at position 1000/1000 */ 
23  accesspoint=VNode.start(NET.tvee[0], nil, 2)
24  air.attach_node(accesspoint.nic[0])
25  accesspoint.nic[0].enable_manet_mode
26  accesspoint.set_position(1000, 1000)
27  /* connect the access point and the router with 100Mbps and 10ms delay */ 
28  TVEE.create_link(router.nic[1], accesspoint.nic[1], 100000, 100000, 10, 10)
29  /* assign IP addresses and routing entries and setup the wireless links */ 
30  NET.autoconf({:wave5data_threshold=>50})
31  /* determine the delay of the path between the phone and the host */ 
32  phone.exec("ping -c 1 #{host.nic[0].ip}")
```

---

### 5.1.2 Graphical Experiment Specification

Our graphical user interface *NETcaptain* [VGR<sup>+</sup>11] constitutes an alternative approach to specify and execute an experiment. The screen shot of *NETcaptain* is shown in Figure 5.3. The base of our GUI is the *Eclipse Rich Client Platform*<sup>28</sup>. The integration into the development platform Eclipse allows for directly evaluating distributed applications using our network emulator.

*NETcaptain* supports an easy, GUI-based specification of the network topology including the link characteristics as well as the assignment of the *Software under Test*. The nodes can be connected by shared media, switched networks or point to point links. To efficiently support large scenarios, importers for common network topology generators (e.g., BRITE [MLMB01]) are integrated. The mobility and connectivity of wireless connected mobile nodes is based on trace files and several radio propagation models (e.g., ray racing [Ste08]). The experiments can be live visualized using a flexible visualization engine and controlled by a powerful scripting engine.

<sup>28</sup><http://www.eclipse.org/home/categories/rcp.php>

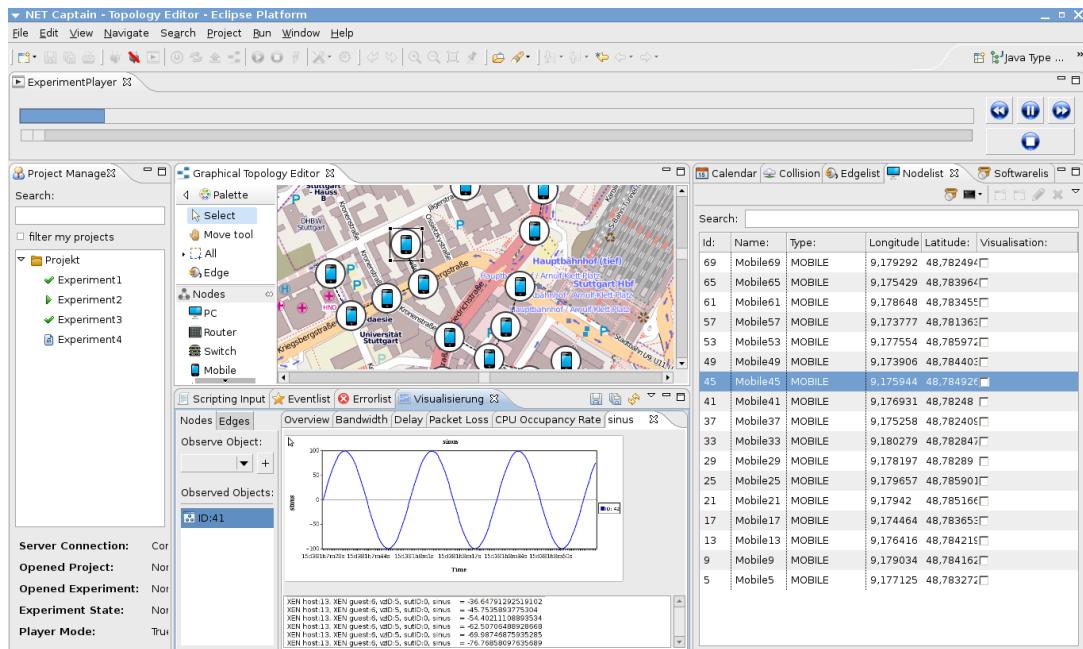


Figure 5.3: *NETcaptain*

## 5.2 Testbed Model

In the following, we present a detailed testbed model for emulation testbeds running at virtual time. The main purpose of the model is to provide a mechanism to calculate the runtime of an experiment based on the experiment specification. The model acts as a foundation of our placement approaches *NETplace* and *NETbalance* (cf. Section 5.3 and Section 5.4, respectively).

A network experiment in NET consists of a set  $\mathbb{N}$  of virtual nodes. Each virtual node  $n_i \in \mathbb{N}$  runs a *SuT* which consumes  $\lambda_i$  CPU cycles and transmits  $\beta_{ij}$  data to virtual node  $n_j$  per time unit. The experiment runs for  $\theta_{\text{virtual}}$  time units of the virtual time. Here, we assume the knowledge of the average data rates produced by the SuT on the links between the virtual nodes. The specification of a virtual node's load and the average data rates is provided by the experimenter or can be gathered during the execution of a scenario. However, the latter approach can only be applied in case the experiment is executed multiple times. Initial work on minimizing the required user interaction to specify this information is provided in the diploma thesis by Zhou [Zho11]. Moreover, the specification of the virtual links' bandwidth provides a worst case estimation of these data rates.

The experiment is executed on a testbed containing a set  $\mathbb{P}$  of physical nodes. Each physical node  $p \in \mathbb{P}$  is equipped with a set  $\mathbb{C}_p$  of CPUs and runs  $|\mathbb{C}_p|$  virtual machines. Each CPU can perform  $\nu_{\text{CPU}}$  CPU cycles per real time unit. We identify each VM by addressing the physical node  $p$  and the CPU  $c \in \mathbb{C}_p$  that is assigned to the VM  $(p, c)$ . The set of VMs is denoted  $\mathbb{V}$ . We define the placement of the virtual nodes onto the virtual machines as a function  $\phi : \mathbb{N} \rightarrow \mathbb{V}$ . Based on this placement  $\phi$ , the cost model  $\mu$ , shown in Equation 5.1, allows for the calculation of a physical node's, say  $p$ , load.

$$\mu : (\phi, p) \mapsto (\Lambda_p^{\text{host-os}}, \Lambda_{p,1}^{\text{vm}}, \dots, \Lambda_{p,|\mathbb{C}_p|}^{\text{vm}}) \quad (5.1)$$

As discussed in Section 4.1.3, there exist approaches to enable communication between virtual machines with and without involving the host operating system (host-os). The presented cost model supports both approaches, however, we focus on host-based network access in the remainder of this section. Due to the fixed assignment of CPUs to the virtual machines and the host operating system, the cost model  $\mu$  determines the load of the host operating system  $\Lambda_p^{\text{host-os}}$  and the load of the virtual machines  $\Lambda_{p,c}^{\text{vm}}$  separately. The unit of  $\Lambda$  is CPU cycles per time unit.

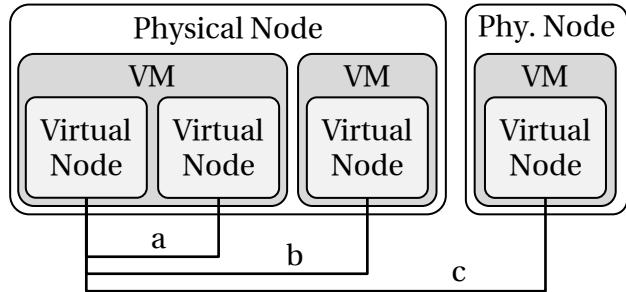


Figure 5.4: Link types: a) intra-vm, b) inter-vm, and c) intra-pnode

In order to define  $\mu$ , we distinguish three types of links between virtual nodes: a) *intra-vm*, b) *inter-vm* and c) *inter-pnode links* (cf. Figure 5.4).

- a) *Intra-vm links* are the most efficient way to connect two virtual nodes. Here, both virtual nodes are running inside the same VM and, therefore, make use of the same CPU. Attaching the virtual nodes' NIC (network interface card) to a software bridge, also running inside the VM, enables communication. Since the communication involves only components inside the VM, only the send ( $VM_{tx}$ ) and receive ( $VM_{rx}$ ) path of the VM's protocol stack is loaded (cf. Table 5.1).
- b) *Inter-vm links* allow for connecting two virtual nodes running in different VMs on the same physical node. Here, communication requires to copy the packets between the VMs and the host-os, which introduces some additional overhead. Evaluations [GHR10] show that inter-vm links are about 10 times more expensive than intra-vm links.
- c) *Inter-pnode links* allow for connecting two virtual nodes running on different physical nodes. Here, packets need to be copied to the host-os and passed down the complete network stack including the device driver for the network hardware. Evaluations [GHR10] show that these type of links introduce the highest overhead and cause about two times more load than the inter-vm links and 20 times more load than the intra-vm links.

Knežević et al. [KSK09] further divide the *inter-vm links* and investigate the overhead when communicating between CPU cores located on the same or on different CPU sockets. Their results confirm our assumption that this differentiation has only a marginal effect on the performance and, therefore, can be neglected.

The last two columns in Table 5.1 show the load generated by the different link types on a physical node (column *pnode*) and the load throughout the testbed (column *Testbed*). The load of a physical node (column *pnode*) is composed by the load of the sending

	VM <sub>tx</sub>	VM <sub>rx</sub>	host-os	pnode	testbed
intra-vm	+	+	-	+	+
inter-vm	++	++	++	++	++
inter-pnode	++	++	+++	++ / ++	+++

Table 5.1: Cost matrix  $\kappa$ , where the number of + denotes the amount of load generated by a link to the system components (no load is indicated by -).

and receiving VM as well as two times the load of *host-os*. In case of inter-pnode links, this load is distributed to two physical nodes. The column *Testbed* defines the generated load throughout the testbed.

Based on the link type, the data rates  $\beta_{ij}$ ,  $\beta_{ji}$  of a link between two virtual nodes  $i$  and  $j$ , and the cost matrix  $\kappa$  (cf. Table 5.1), Equation 5.2 defines the generated load of the host-os  $\Lambda_p^{\text{host-os}}$ .

$$\Lambda_p^{\text{host-os}} = \sum_{\substack{n_i, n_j \in \mathbb{N} \\ \phi(n_i) = (p, c) \\ \phi(n_j) = (p', c')}} (\beta_{ij} + \beta_{ji}) \cdot \begin{cases} \kappa_{\text{intra-vm, host-os}} & \text{if } p = p' \wedge c = c', \\ \kappa_{\text{inter-vm, host-os}} & \text{if } p = p' \wedge c \neq c', \\ \kappa_{\text{inter-pnode, host-os}} & \text{if } p \neq p' \end{cases} \quad (5.2)$$

The load of a virtual machine  $\Lambda_{p,c}^{\text{vm}}$  is calculated using Equation 5.3, where  $\Lambda_{p,c}^{\text{VM}_{\text{tx}}}$  and  $\Lambda_{p,c}^{\text{VM}_{\text{rx}}}$  are defined analogous to  $\Lambda_p^{\text{host-os}}$ . The virtual machine running the virtual node  $i$  that transmits data is loaded by  $\kappa_{\text{VM}_{\text{tx}}}$  and the virtual machine running the virtual node  $j$  that receives data is loaded by  $\kappa_{\text{VM}_{\text{rx}}}$ . The SuT additionally loads the virtual machines by  $\lambda_i$  and  $\lambda_j$ , respectively.

$$\Lambda_{p,c}^{\text{vm}} = \Lambda_{p,c}^{\text{VM}_{\text{tx}}} + \Lambda_{p,c}^{\text{VM}_{\text{rx}}} + \sum_{n_i \in \mathbb{N} \wedge \phi(n_i) = (p, c)} \lambda_i \quad (5.3)$$

We calculate the load of the maximum loaded physical CPU  $\Lambda_p^{\text{max}}$  of physical node  $p$  using Equation 5.4. Since the virtual CPUs of the virtual machines are pinned to physical CPUs, a physical CPU  $(p, c)$  experiences at least the load  $\Lambda_{p,c}^{\text{vm}}$  of the virtual machine that it is assigned to. Therefore, the maximum loaded CPU experiences at least the load of the maximum loaded virtual machine. In addition, the load  $\Lambda_p^{\text{host-os}}$  of the host-os can be distributed arbitrarily to the CPUs. Therefore, each CPU experience at least the

fraction  $\frac{1}{|\mathbb{C}_p|}$  of the overall load generated on the physical node  $p$ .

$$\Lambda_p^{\max} = \max \left( \max_{c' \in \mathbb{C}_p} \left( \Lambda_{p,c'}^{\text{vm}} \right), \frac{1}{|\mathbb{C}_p|} \cdot \left( \Lambda_p^{\text{host-os}} + \sum_{c' \in \mathbb{C}_p} \Lambda_{p,c'}^{\text{vm}} \right) \right) \quad (5.4)$$

Knowing the load of the maximum loaded CPUs of each physical node, we can define the experiment runtime using Equation 5.5. The maximum loaded CPU throughout the testbed executes  $\max_{p \in \mathbb{P}} (\Lambda_p^{\max}) \cdot \theta_{\text{virtual}}$  CPU cycles during the experiment running for  $\theta_{\text{virtual}}$  time units (in virtual time). The number of cycles divided by the speed of the CPUs  $v_{\text{CPU}}$  results in the experiment runtime  $\theta_{\text{real}}$  (in real time).

$$\theta_{\text{real}} = \max_{p \in \mathbb{P}} \left( \Lambda_p^{\max} \right) \cdot \frac{\theta_{\text{virtual}}}{v_{\text{CPU}}} \quad (5.5)$$

The cost matrix  $\kappa$  highly depends on the used hardware and software base, such as the speed of the physical memory, the CPU architecture, the VM implementation, and the used operating system. Therefore, we propose the following approach to determine the cost matrix  $\kappa$  for a testbed: First, a sample scenario is executed with a number of arbitrary placements while we monitor the generated load on the VMs and the host-os, and the data rates. Second, genetic programming is used to find values for the cost matrix  $\kappa$  in Table 5.1 by minimizing the difference between the measured load and the calculated load based on  $\kappa$  and the measured data rates. During the evaluation of the testbed model in Section 5.5.1, we provide details on the used scenarios to determine the cost matrix  $\kappa$ .

## 5.3 Initial Node Placement

The placement of virtual nodes onto physical nodes is essential to achieve good emulation performance. Different emulation architectures require different assignment strategies. In this section, we first discuss related network emulators and their placement strategies. Second, we provide a formal problem statement. Finally, we introduce our placement approach *NETplace* [GHR10] to minimize the runtime of experiments executed by our node and time virtualized network emulator. A detailed evaluation of the initial node placement is presented in the Section 5.5.2.

### 5.3.1 Related Work

In real time testbeds [AC06, HRS<sup>+</sup>08] the capacities of network links in terms of available bandwidth and the processing capabilities of the physical nodes are limited. In testbeds based on real-world networks [CCR<sup>+</sup>03] the delay of the physical links between the nodes constrains the assignment of virtual nodes as well. There exist several approaches to solve this *Constraint Satisfaction Problem* (CSP) [And02].

Evolutionary algorithms are used in Emulab's *assign* [RAL03, ROLV06] and the placement strategy by Liu et al. [LLXC05]. While *assign* uses simulated annealing, the latter makes use of a genetic algorithm to solve the CSP. In Virtual Grid [KLH<sup>+</sup>05] the CSP is mapped to a multidimensional range search. The idea of *wanassign* [CBMP04] is to model the testbed properties as an *interference matrix* and the requirements of links between virtual nodes as a *constraint matrix*. The CSP is solved by searching for an embedding of the *constraint matrix* to the *interference matrix* using a backtracking approach with a heuristics to prune the search tree and caching of partial solutions. In *EMPOWER* [ZN03] the capacity of each physical node represents the network load that a physical node can handle. Assuming the knowledge of the generated data rates of the virtual nodes, a *Bin Packing* algorithm based on a heuristic best-fit strategy is used to map the virtual nodes onto the physical resources. McGeer et al. [MAS10] use graph partitioning techniques to assign nodes to switches without exceeding the inter-switch capacity. In contrast to our strategy, all these approaches do not distinguish the costs of the different link types. In addition, every solution satisfying the constraints has the same experiment runtime. Therefore, these approaches can hardly be used to minimize the experiment runtime of a virtual time-based network emulator.

In parallel computing [HK00] as well as in network simulation [LC04] and network emulation [YED<sup>+</sup>03] the execution time is minimized by modeling networks of com-

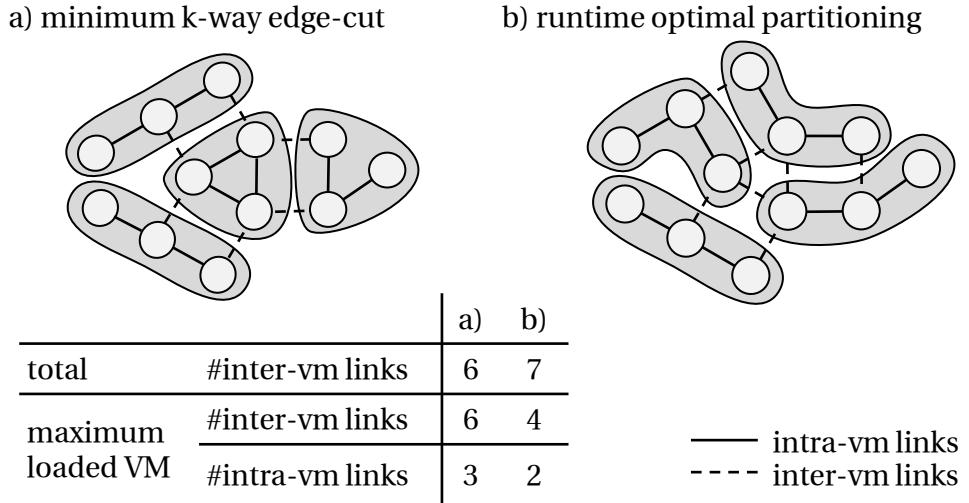


Figure 5.5: Network with 12 virtual nodes partitioned into 4 partitions (1 physical node with 4 VMs). K-way edge-cut (a) minimizes the total number of inter-vm links. However, the maximum loaded CPU (VM) determines the experiment runtime. b) shows an optimal partitioning minimizing the load of the maximum loaded VM and, thus, the runtime of an experiment.

ponents or virtual nodes as graphs. The weight of an edge represents the amount of traffic between the virtual nodes, whereby the weight of a vertex denotes the amount of computation. Frameworks like *metis* [KK98a] can partition the graph in subgraphs while minimizing the summed weight of the edges between partitions (edge-cut) and at the same time balancing the summed vertex weights in each partition (*minimized k-way edge-cut*). However, this approach has several problems which prohibit the minimization of the runtime of network emulation experiments: First, the cut edges are not balanced between the partitions. Since inter-vm and inter-pnode links generate large CPU load, the load is unequally distributed to the physical nodes (cf. Figure 5.5). Second, this approach does not consider multiple VMs per physical node. Inter-vm and inter-pnode links generate a different amount of costs and, therefore, minimizing only the total edge-cut may result in suboptimal runtime. Third, multi-core architectures are not considered. Depending on the link type (intra-vm, inter-vm and inter-pnode) load is generated in different system components which run on different CPU cores. Finally, load generated by links inside a partition (intra-vm links) is also not considered by edge-cut-based approaches.

### 5.3.2 Formal Problem Statement

Before presenting our placement approach, we first define the problem formally:

**Placement Problem:** *Given a network experiment with virtual nodes  $\mathbb{N}$  and an emulation testbed consisting of physical nodes  $\mathbb{P}$  that execute the virtual machines  $\mathbb{V}$  (for details see Section 5.2), we are searching for a function  $\phi: \mathbb{N} \rightarrow \mathbb{V}$  that minimizes  $\max_{p \in \mathbb{P}} (\Lambda_p^{\max})$ .*

Several related placement problems [WLS<sup>+</sup>02, VYW<sup>+</sup>02, RAL03, LC03, CBMP04, HRS<sup>+</sup>04, LLXC05, GRL05, AC06] are proofed to be NP-hard. As outlined during the discussion of the related work, our placement problem has a different focus and, thus, related works' proofs of the NP-hardness cannot be directly applied to our problem. Therefore, we give a proof of the NP-hardness of our placement problem by reducing the NP-complete problem of *Bin Packing* [GJ79] to our *Placement Problem*.

*Bin Packing* is defined as follows: Given  $k \in \mathbb{N}^+$  bins with a capacity  $c \in \mathbb{N}^+$  and  $n \in \mathbb{N}^+$  objects with sizes  $a_1, a_2, \dots, a_n$ , where  $a_i \leq c$ . The question is  $\exists f: 1..n \Rightarrow 1..k$ , so that  $\forall j=1..k \left( \sum_{f(i)=j}^{i=1..n} a_i \right) \leq c$ .

To reduce *Bin Packing* to our placement problem we construct a network experiment with  $n$  virtual nodes, where virtual node  $n_i$  runs a SuT with a load of  $a_i$  CPU cycles per time unit. We create a fully connected network with  $\beta_{ij} = 1$ . The virtual network is placed onto a testbed consisting of one physical node  $p$  equipped with  $k$  CPUs running  $k$  virtual machines and using a cost matrix  $\kappa = 0$ . A solution of *Bin Packing* exists iff the load of the most loaded VM  $\Lambda_p^{\max}$  is less or equal than  $c$ .  $\square$

### 5.3.3 Placement Algorithms

Next, we present our approach to calculate the placement  $\phi$  that minimizes the experiment runtime. As shown in the previous section, the placement problem is NP-hard and, therefore, rather than calculating the optimal solutions we rely on heuristics to quickly calculate a reasonable good solution. In the following, we first propose two extensions to the original edge-cut algorithm to overcome its shortcomings. As an alternative, we propose a simple greedy algorithm to calculate the placement. After the placement calculation, a subsequent optimization phase further reduces the runtime of the experiment.

#### Edge-Cut-based Approaches

In order to place virtual nodes using edge-cut-based approaches, the virtual network is modeled as a weighted graph  $\gamma$ . The weight of a vertex  $v_i$ , which represents a virtual node, is defined as the load of the virtual nodes' SuT  $\lambda_i$  and the weight of an edge  $e_{ij}$  is defined as the bandwidth of the virtual link  $(\beta_{ij} + \beta_{ji})$ . The edge-cut algorithm [KK98a] is used to partition the graph into  $n$  partitions, where  $n$  is the number of virtual machines in the testbed. The nodes of each partition are placed onto the same virtual machine. In the following, we extend this approach to minimize the runtime of network emulation experiments.

**Balanced Edge-Cut  $E_B$**  In order to consider intra-vm links, we add the cost of emulating the intra-vm links to the vertex weight. The vertex weight  $v_i$  is, therefore, redefined as the load of the virtual nodes' SuT  $\lambda_i$  plus the sum of all virtual links of the virtual node times the intra-vm costs (cf. Equation 5.6).

$$v_i = \lambda_i + \sum_{n_j \in \mathbb{N}} (\beta_{ij} + \beta_{ji}) \cdot \kappa_{\text{intra-vm,pnode}} \quad (5.6)$$

Since the edge-cut algorithm balances the vertex weights between partitions, costs generated by the emulation of intra-vm links do not cause load imbalances between physical nodes. However, inter-vm and inter-pnode links can still cause load imbalances. From now on, we assume the modified vertex weights.

**Hierarchical Edge-Cut  $E_H$**  In order to support multiple VMs per physical node and minimizing the inter-pnode links, algorithm  $E_H$  partitions the graph  $\gamma$  two times. During

the first run, we partition the graph  $\gamma$  into  $|\mathbb{P}|$  partitions  $\{\gamma_1, \dots, \gamma_{|\mathbb{P}|}\}$ . Each partition is assigned to a single physical node. For each physical node  $p$ , we again partition the corresponding subgraph  $\gamma_p$  into  $|\mathbb{C}_p|$  partitions  $\{\gamma_{p,1}, \dots, \gamma_{p,|\mathbb{C}_p|}\}$ . The partitions  $\gamma_{p,c}$ , generated in this second run, are assigned to the virtual machines running on the physical node  $p$ .

### Greedy Approach

A simple greedy approach  $G$  constitutes an alternative to place virtual nodes. Here, the assignment of virtual nodes to clusters (virtual nodes running inside the same VM) consists of two phases. First, we assign one random initial virtual node to each cluster. Second, we assign the remaining nodes randomly, one by one to the minimum loaded cluster.

Selecting the minimum loaded CPU or virtual machine requires the calculation of their load. For efficiency, we update the load on these components incrementally after assigning a virtual node to a cluster. When updating the load of the components, we need to handle the links between an already assigned virtual node  $i$  and an unassigned virtual node  $j$ . The type of these links depends on the future assignment of  $j$ . Therefore, we propose a heuristics to estimate the load generated by these links. Following an optimistic approach, we consider only those links, where both virtual nodes are already placed. This model reflects the actual load of the intermediate assignment (unassigned nodes are temporarily removed).

To select the minimum loaded cluster, we first determine the physical node  $p$  where the load of the maximum loaded CPU  $\max_{c \in \mathbb{C}_p} (\Lambda_{p,c})$  is minimal. From the VMs running on this physical node, we select the VM  $(p, c)$  where the load  $\Lambda_{p,c}^{\text{vm}}$  is minimal. Assigning additional virtual nodes to this VM will unlikely increase the experiment time.

Due to the random selection of the virtual nodes, the greedy approach balances the load between the physical nodes without minimizing the inter-vm and inter-pnode links. Therefore, we propose to optimize the placement in a subsequent optimization phase. This optimization is used for the greedy and the edge-cut-based approach.

### Optimization of Node Placements

Due to the use of heuristics, the introduced clustering algorithms cannot guarantee that the load of the maximum loaded CPU is minimized. Therefore, an optimization is

performed after the cluster creation to reduce the load on the maximum loaded CPU, which decreases the experiment runtime. The proposed optimization is performed after the edge-cut based approaches as well as the greedy approaches.

Let  $\{t_1, \dots, t_{|\mathbb{V}|}\} = a$  be a placement of virtual nodes  $\mathbb{N}$  onto VMs  $\mathbb{V}$ , where  $t_i \cap t_j = \emptyset$  and  $\bigcup_{i=1}^{|\mathbb{V}|} t_i = \mathbb{N}$ . We also call such a placement a *state*. In order to optimize a placement, we use hill climbing to minimize a cost function  $\zeta(a)$ . We define  $\zeta$  later in this section. During each round, we generate *neighboring states*  $\{a'_1, \dots, a'_e\}$  by removing a virtual node  $n$  from a cluster  $i$  and assigning  $n$  to a cluster  $j$ :  $\{t_1, \dots, t_i \setminus \{n\}, \dots, t_j \cup \{n\}, \dots, t_{|\mathbb{V}|}\}$ . The *neighboring states* are rated by  $\zeta(a)$ . In case the costs of the best *neighboring state*  $a'_b$  is lower than the costs of the current state  $\zeta(a'_b) < \zeta(a)$  the optimization continues with the state  $a'_b$ .

Due to the large number of neighboring states  $O(|\mathbb{N}| \cdot |\mathbb{V}|)$ , rating these states is expensive. In order to reduce this effort, we propose to sequentially generate and rate the neighboring states. Instead of generating all the states in each optimization round we generate and evaluate the neighboring states iteratively  $a'_i \in \{a'_1, \dots, a'_e\}$ . In case  $\zeta(a'_i) < \zeta(a)$  the optimization continues with  $a'_i$ , otherwise we check the next neighboring state of  $a$ . The optimization terminates if we cannot find any neighboring state with lower costs or a time limit is reached. However, experiments have shown that the local optimum was always reached after a short period of time.

The number of neighboring states can be further reduced by filtering the neighboring states using a heuristics. We remove states from the set of neighboring states which will unlikely have lower costs than the current state. In order to minimize the costs, the number of inter-pnode and inter-vm links must be minimized. Assigning a virtual node  $i$  to a cluster  $c$ , which runs no virtual node  $j$  connected to  $i$  ( $\forall j \in c : \beta_{ij} = 0 \wedge \beta_{ji} = 0$ ), will unlikely reduce the costs because all virtual links of the virtual node  $n$  will become inter-vm or inter-pnode links. Therefore, we only consider neighboring states  $\{t_1, \dots, t_i \setminus \{n\}, \dots, t_j \cup \{n\}, \dots, t_{|\mathbb{V}|}\}$ , where  $\exists m \in t_j : \beta_{nm} > 0 \vee \beta_{mn} > 0$ . However, evaluation shows that this heuristics is too restrictive. Therefore, instead of removing all affected states from the set of neighboring states, we include some randomly selected states to the set of neighboring states. This approach allows for exploring more states before reaching a local minimum.

Based on our primary optimization goal, an obvious cost function would be the experiment runtime of an assignment. However, this cost function has a lot of plateaus, because the runtime of an assignment is only decreased in the case where the load of the maximum loaded CPU  $\max(\Lambda_{p,c})$  is reduced. The hill climbing-based optimization

cannot escape from such a plateau because the gradient is zero in all directions. Therefore, we propose a two-part cost function (cf. Equation 5.7) to eliminate the plateaus. The first part of the cost function is determined by the maximum loaded CPU  $\max(\Lambda_{p,c})$ . The second part is calculated by summing up the squared load of all physical CPUs. We are using the squared load of a CPU because, this metric penalizes assignments with unequally loaded CPUs.

$$\zeta = \begin{pmatrix} \zeta_1 \\ \zeta_2 \end{pmatrix} = \begin{pmatrix} \max_{p \in \mathbb{P} \wedge c \in \mathbb{C}_p} (\Lambda_{p,c}) \\ \sum_{p \in \mathbb{P} \wedge c \in \mathbb{C}_p} (\Lambda_{p,c})^2 \end{pmatrix} \quad (5.7)$$

Equation 5.8 is used to compare a state  $a$  and a neighboring state  $a'$ . Here, we first compare  $\zeta_1$  because reducing the load of the maximum loaded CPU results directly in a reduced experiment runtime. In case  $\zeta_1$  is equal for both states (third line of Equation 5.8), we compare  $\zeta_2$ .

$$\zeta(a') < \zeta(a) \Leftrightarrow \begin{cases} \text{true} & \text{if } \zeta_1(a') < \zeta_1(a), \\ \text{false} & \text{if } \zeta_1(a') > \zeta_1(a), \\ \zeta_2(a') < \zeta_2(a) & \text{if } \zeta_1(a') = \zeta_1(a) \end{cases} \quad (5.8)$$

## 5.4 Dynamic Reconfiguration

Varying CPU or network load of virtual nodes during an experiment run may result in a temporal suboptimal placement and, thus, a longer experiment runtime. In this section, we present *NETbalance* [GHR11] an extended approach to minimize the experiment runtime in scenarios with varying resource requirements. At runtime, *NETbalance* detects changes in the resource requirements of virtual nodes and recalculates an optimized placement of the virtual nodes. We adopt the concept of *live migration* [CFH<sup>+</sup>05] to transform the current placement into the optimized placement by migrating virtual nodes between the virtual machines.

The remainder of this section is structured as follows: First, we present related work in the field of migration and load balancing. Second, we extend our emulation architecture to efficiently support the migration of virtual nodes. Third, we introduce a cost model covering the migration costs. Finally, we discuss the concepts of placement adaptation. A detailed evaluation of the reconfiguration concepts is presented in Section 5.5.3.

### 5.4.1 Related Work

Live migration of virtual machines [MKK08, Han09, ASR<sup>+</sup>10] as well as process migration [DO91, HH02, BD02, OSSN02] is commonly available in virtualization products and operating systems, respectively. These migration techniques are used in a wide area ranging from the field pervasive computing [KS02], where mobile applications are migrated between users' devices, to *High Performance Computing* (HPC) to build fault tolerant systems [RHV06, RH11]. Up to now, there has been no network emulator that uses the migration of virtual nodes to reduce the runtime of experiments. Therefore, we investigate approaches from other areas using similar concepts for their applicability to our problem.

In data centers live migration of virtual machines is used to avoid load imbalances [AK10] and to eliminate load hot spots [KBKK06] by migrating virtual machines from high loaded servers to servers with low load. These approaches consider load of different resources: CPU [KBKK06, WSVY07], memory [WTLS<sup>+</sup>09], network [SGRC10], and storage [SKM08]. A multidimensional load, covering multiple resources, is often considered [SKM08, SGRC10]. These systems are using centralized components [KBKK06, WTLS<sup>+</sup>09, AK10] or distributed approaches [SGRC10] to select the VM to be migrated. However, in contrast to our system, VMs can be migrated without suspending all other VMs in the data center.

The migration of virtual nodes is similar to task migration in parallel computing, too. Here, load balancing [SKS02, WLR02] is achieved by migrating tasks from nodes with high load to nodes with low load. Willebeek-LeMair and Reeves [WLR02] investigate several different algorithms (*GM, SID, RID, HBM, DEM*) for minimizing the computation and communication effort of identifying the highly loaded nodes. In contrast to our problem, local information can be used to decide which task to migrate, because the placement of one task does not influence the execution costs of another task. Additionally, the migration of tasks can be performed independently without the need to suspend all other task in the system.

In the area of process migration and virtual machine migration, there exist several approaches for minimizing the migration time. Techniques such as *pre-copy* [TLC85, CFH<sup>+</sup>05], *demand-migration/post-copy* [Zay87, HDG09], or combinations [LZW<sup>+</sup>08] minimize the time between suspending and resuming a process. The idea is to transfer the memory state before suspending or after resuming the process and, therefore, to minimize the time a service (provided by the process) is unaccessible. However, applied to network emulation, in both approaches the state is transferred in parallel to the running experiment and, therefore, will increase the CPU usage which leads to a slower experiment execution. Additionally, the total amount of transferred data is increased, since the memory is transferred multiple times or memory pages need to be explicitly requested from the source.

Other techniques such as *zero elimination* [SCP<sup>+</sup>02], *compression* [SCP<sup>+</sup>02] and *duplicate elimination* [SCP<sup>+</sup>02] can be used to reduce the size of the transferred state. However, these approaches introduce additional CPU overhead. In the field of grid computing application support is used to reduce the memory state. Instead of complete operating systems or processes, here, only lightweight threads are migrated. However, this approach limits the applicability of network emulation because of the required adaptation of *Software under Test* to support migration.

In the field of network emulation, migration has been used to increase the scalability for performance evaluation of wireless connected applications in heterogeneous testbeds [HLW<sup>+</sup>11]. In such testbeds some testbed nodes are connected to an emulated RF environment (*Radio Frequency*). Due to the limited capacity of these nodes, disconnected virtual nodes (no radio connection in the virtual topology) are migrated to testbed nodes without access to the emulated RF environment. However, since the virtual nodes are disconnected from the virtual topology, here, the migration can run in parallel to the running experiment.

Related to the migration of virtual nodes is creating snapshots of virtual nodes [SPYH03]. Emulab has been extended to support preemptively swap out running experiments [BRHL09]. Experiments can be resumed from these snapshots. In contrast to our system, here, the focus is not on runtime minimization but on offline analysis and debugging of the *Software under Test*.

### 5.4.2 Migration Support for Network Emulation

An integral part of *NETbalance* is the replacement of virtual nodes using live migration [CFH<sup>+</sup>05]. To ensure that the replacement of virtual nodes does not influence the emulation results, the migration of virtual nodes must be transparent to the SuT. Transparency requires to perform changes of the placement during a suspended experiment because otherwise the SuT could detect the migration of virtual nodes. However, this experiment interruption contradicts with the goal of reducing the runtime of the network experiments and, therefore, the amount of time when the experiment is suspended needs to be minimized. In the following, we present an extension to our emulation architecture (cf. Section 4.1), to efficiently and transparently migrate virtual nodes.

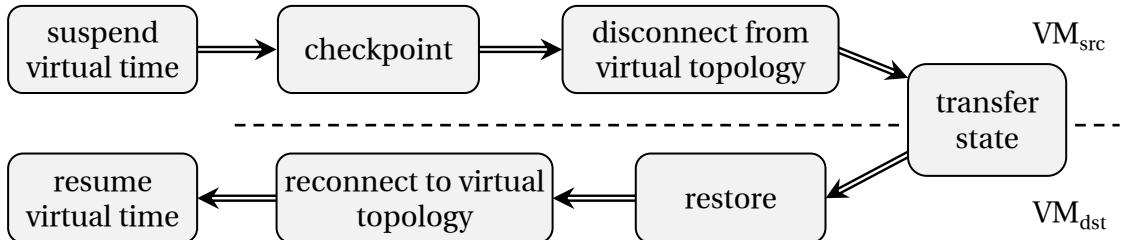


Figure 5.6: Transparent migration of virtual nodes

The process to migrate virtual nodes transparently is visualized in Figure 5.6. In the following, we explain each step in detail. In order to achieve transparency, we suspend the experiment synchronously on all physical nodes which includes two phases. First, by setting the time dilation factor to infinity, the virtual clocks are paused. This ensures that *NETshaper* will not deliver any frames and that timed actions are not triggered, e.g., in the protocol stack. In the second phase, we exclude the processes of the virtual nodes from process scheduling.

After the experiment is suspended, we change the placement of virtual nodes. For this, we adopt the concepts of the *ZAP* system [OSSN02]. First, we create a snapshot of a virtual node using checkpointing. Figure 5.7 shows the state of a virtual node for each

	VM <sub>src</sub>	VM <sub>dst</sub>
Application Layer	SuT	→
Transport Layer	Sockets	→
Network Layer	Routing tables	→
Data Link Layer	NETshaper	→
Virtual Topology	Bridges	→
	VLANs	→

Figure 5.7: State of a virtual node

protocol layer. The *Application* layer state contains the memory pages and open file descriptors of the SuT, the *Transport* layer state contains the open sockets and the state of the corresponding protocols, and the *Network* layer state contains the IP addresses as well as the routing tables. We extended the state of the *Data Link* layer by the state of *NETshaper*, including buffered messages. After disconnecting the virtual node from the virtual topology, the virtual node is shut down. The state of the virtual node is then transferred to VM<sub>dst</sub> and restored thereafter. RDMA<sup>29</sup>-based communication [HGLP07, XOP11], where the data transfer between physical nodes is offloaded to the network devices, can be used to speed up the state transfer. The network interfaces of the restored virtual nodes are reattached to the virtual topology.

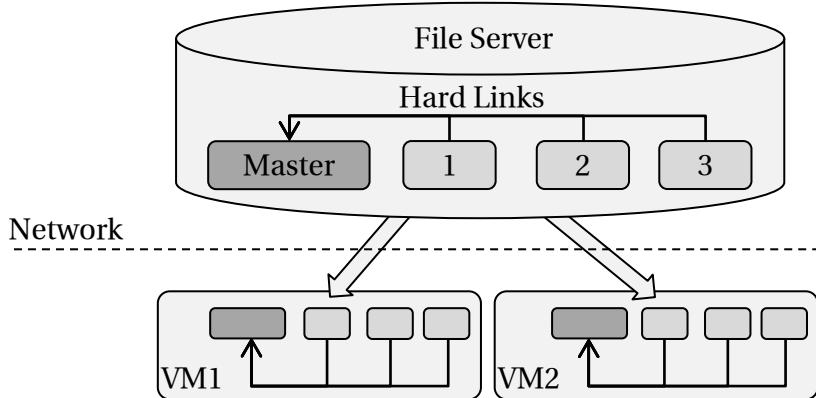


Figure 5.8: Server-based file system of virtual nodes

The SuT might have modified the file system or it might have open file descriptors. Therefore, we need to transfer the virtual node's file system to VM<sub>dst</sub>. Due to the typical size of a file system, copying all files introduces a large overhead. To avoid this overhead,

<sup>29</sup>remote direct memory access

we store the file system of a virtual node on a central server<sup>30</sup> (cf. Figure 5.8). Typically, most files of a virtual node (including the system files of the operating system and the libraries of the SuT) are read-only and shared among the virtual nodes. Therefore, all virtual nodes use a common *Copy-on-Write* [FG95] file system to share these files. To minimize the overhead, we are using hard links to make shared files available to all virtual nodes. This approach saves a lot of disk space on the file server and shared files need to be cached only once on the file server and the virtual machines.

The caching effort to keep the entire file system in memory is almost independent of the number of virtual nodes. Node-specific files are only cached by the VM running the virtual node. Buffering of write operations and caching of read operations hide the latencies of the network-based file I/O. Due to the concept of the file server, the effort of synchronizing the virtual nodes' file system is limited to writing back the modified files to the file server. Using techniques such as NFS over RDMA [NCTP07], the CPUs of the physical nodes are not involved in the file transfer. The files are written back while the virtual nodes are suspended in parallel to the reconfiguration process. Since we are assuming only small changes to the file system, implying a fast synchronization and a negligible effect on the node's state size, the synchronization time does not contribute to migration time of a virtual node state. In the case of larger changes, the state of the virtual node grows. In our evaluation, we show the impact of the state size on the performance of *NETbalance*.

The migration is completed by resuming the execution of all virtual nodes and by restoring the time dilation factor. From the SuT's point of view the migration takes zero time. Therefore, the migration is not detectable by the SuT.

### 5.4.3 Migration Cost Model

The time to migrate virtual nodes counteracts the possible experiment runtime reduction of a placement reconfiguration. In order to calculate if the increased emulation speed outweighs the reconfiguration time  $T_r$ , we need a cost model for the reconfiguration time. Since we are using suspend/resume migration [KS02], the reconfiguration time  $T_r$  is defined as follows:

$$T_r = T_{\text{suspend}} + T_{\text{migrate}} + T_{\text{change-topology}} + T_{\text{resume}} \quad (5.9)$$

---

<sup>30</sup>The central file server could be implemented by a cluster of file servers to avoid performance bottlenecks

$T_{\text{suspend}}$  and  $T_{\text{resume}}$  are small, because we only need to exclude or include the virtual nodes from process scheduling. The time for changing the virtual topology is short, too. The dominating factor of  $T_r$  is  $T_{\text{migrate}}$ , because it grows linearly with the memory pages allocated by the SuT. The actual values for  $T_{\text{suspend}}$ ,  $T_{\text{migrate}}$ ,  $T_{\text{change-topology}}$ , and  $T_{\text{resume}}$  can be measured based on a sample scenario (cf. evaluations in Section 5.5.3). Additionally, better estimations can be learned while the experiment is running.

The migrations of virtual nodes running in the same Virtual Machine (VM) are performed sequentially. However, since the migration of a virtual node generates only load on  $\text{VM}_{\text{src}}$  and  $\text{VM}_{\text{dst}}$ , we can migrate virtual nodes running on different VMs in parallel. The reconfiguration costs are calculated for each VM based on the migrations involving the VM. The VM with the maximum value of  $T_r$  determines the overall reconfiguration costs.

#### 5.4.4 Reconfiguration Concepts

In this section, we discuss the calculation of a new placement of virtual nodes that minimizes the remaining experiment runtime. After changing the placement, the experiment runs with an increased execution speed. However, this speedup only leads to a reduction in the runtime if it outweighs the time required for migrating the virtual nodes. The time for which the experiment can run with the increased speed after a reconfiguration determines the overall speedup. Our assumption is that we can predict the future load accurately within a certain time period. We call this time period the *prediction window*. Based on the prediction window and the migration costs, we can determine if the migration of virtual nodes reduces the experiment runtime. In the following, we discuss the prediction window and the algorithm for optimizing the placement in detail.

The research on load prediction shows, that the load of a machine can be predicted up to 30 s in advance [DO00]. As reported by Yang et al. [YFS03] a very simple load predictor using the last measured value as the prediction gives similar results to more sophisticated approaches. In order to minimize the computation effort, we apply this simple prediction scheme. Due to the usage of virtual time, the changes of a virtual node's load experience time dilation. Therefore, the prediction window  $T_p$  is scaled by the time dilation factor  $\tau$ , and we can assume the load to be known for a real time window of  $T_p \cdot \tau$ .

The load of the virtual nodes is captured by a load monitor running inside the VMs and

periodically sent to the coordinator with an interval equal to the prediction window. Even for large scenarios with a thousand virtual nodes per physical node, the amount of data is about 20 kB per physical node<sup>31</sup>. Significant changes in the load of virtual nodes trigger the calculation of a new placement  $\phi'$ . More precisely, if the actual TDF of the experiment deviates from the TDF calculated on the basis of the load reports, the coordinator can initiate an optimization of the placement. To calculate  $\phi'$ , the coordinator adapts the current placement  $\phi$  to the changed load. Using the *testbed cost model* (cf. Section 5.2) developed for *NETplace* [GHR10], we can calculate the time dilation factor for the current placement  $\tau_\phi$  and the new placement  $\tau_{\phi'}$ . For the transition  $\phi \rightarrow \phi'$ , we need to migrate virtual nodes. This migration requires reconfiguration costs  $T_r$ , that can be calculated using our *migration cost model* introduced in the previous section.

Since we can predict the load of the virtual nodes for the time window  $T_p$ , we limit the optimization to the time  $T_o$  with  $T_o \ll T_p$ . Note that similar to  $T_p$ , the optimization time  $T_o$  is scaled by the TDF  $\tau$ . After  $T_o \cdot \tau$ , we abort the simulated annealing-based algorithm [Bar11] used for minimizing the cost function  $\chi$ :

$$\chi = [(T_p - T_o) \cdot \tau_{\phi'} + T_r] - (T_p - T_o) \cdot \tau_\phi \quad (5.10)$$

$\chi$  represents the alteration of the experiment runtime in the prediction window  $T_p$ . The runtime of the current placement  $\phi$  is subtracted from the runtime of the new placement  $\phi'$ , taking into account the time  $T_r$  required for the reconfiguration and the different time dilation factors. Since we need time  $T_o$  for calculating  $\phi'$  and for executing the transformation  $\phi \rightarrow \phi'$ ,  $\phi'$  takes effect over the time window  $T_p - T_o$ . If  $\chi$  is negative, then the transition to  $\phi'$  will result in a speedup of the experiment and *NETbalance* configures the system accordingly. If, however,  $\chi$  is positive, then  $\phi'$  performs worse than  $\phi$  and we keep the configuration  $\phi$ . Thus, the experiment runtime cannot increase through the optimization.

The value of  $T_o$  determines the performance of *NETbalance*. A larger  $T_o$  increases the time for finding better placements. At the same time, however, the time  $T_p - T_o$  left for actually running the better configuration  $\phi'$  gets smaller. In our evaluation, we investigate the optimal value of  $T_o$ .

Small changes of a virtual node's load may result in slightly different optimal placements and, therefore, in a potential for oscillation. However, the gain of a new placement has

---

<sup>31</sup>Scenario with four network links per virtual node

to exceed the reconfiguration costs; otherwise, it is discarded. This effectively serves as a hysteresis, avoiding constant reconfiguration with minimal gain.

After calculating a new placement of virtual nodes, we need to enforce the changes to the placement by migrating virtual nodes using the techniques discussed in Section 5.4.2.

## 5.5 Evaluation

The evaluation of the performance of the experiment configuration is structured as follows. First, we show the accuracy of the testbed model. Second, we discuss the efficiency and effectiveness of the initial node placement (*NETplace*). Finally, we evaluate the performance of the dynamic reconfiguration (*NETbalance*).

### 5.5.1 Testbed Model

As the basis for the evaluation of the testbed model we used the new *NET* cluster (cf. Section 3.1). Each cluster node is equipped with two QuadCore Intel Xeon processors running at 2.4 GHz and 16 GB of main memory (RAM). The nodes are interconnected by a 1 Gbps for control network and 10 Gbps emulation network. Depending on the number of VMs used for the experiment, we deactivate non assigned CPUs.

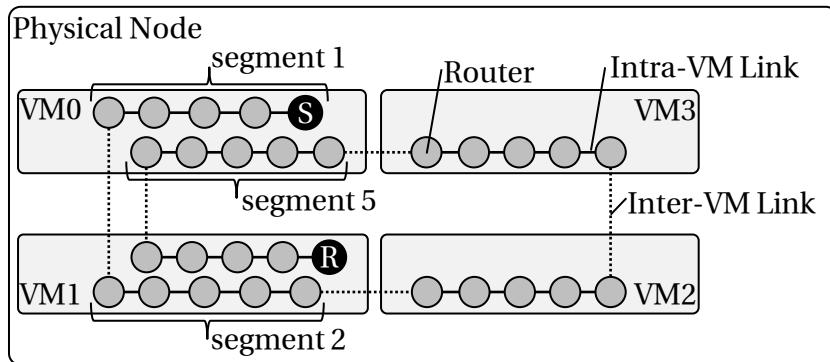


Figure 5.9: Scenario to determine testbed's cost matrix (Router chain with 6 segments, each of length 5)

In order to evaluate the accuracy of the cost model, we determine the cost matrix  $\kappa$  using the method introduced in Section 5.2. A router chain loaded by a single TCP connection (cf. Figure 5.9) is used as the sample scenario. The chain is divided into several equal-sized segments. All virtual nodes (routers) of the same segment are placed on the same virtual machine. The segments are alternately placed on the virtual machines. By varying the segment length and the number of segments, all ratios of intra-vm and inter-vm links are possible. In order to get a setup with inter-pnode links, we replace the inter-vm links with inter-pnode links, by inserting an additional virtual node between each neighboring pair of segments. These additional nodes are placed onto a second physical node.

	VM <sub>tx</sub>	VM <sub>rx</sub>	host-os	pnode	testbed
intra-vm	0.43	0.01		0.88	0.88
inter-vm	1.11	1.09	3.06	8.32	8.32
inter-pnode	2.01	1.67	5.67	7.68/7.31	15.09

Table 5.2: Cost matrix  $\kappa$  for an emulation testbed consisting of quad-core machines (unit of  $\kappa$  is  $\frac{\text{cycles}}{\text{byte}}$ )

Table 5.2 shows the cost matrix  $\kappa$  for the three types of links. The values indicate the number of required CPU cycles to transmit one byte. Column  $\text{VM}_{\text{tx}}$  shows the costs of a VM running a virtual node that transmits data. Column  $\text{VM}_{\text{rx}}$  shows the costs of a VM running a virtual node that receives data. The costs of the host-os are shown in Column *host-os*. Column *pnode* indicates the summed up costs per physical node (costs for the VMs and the host-os). In case of an inter-pnode link, we provide the costs for the physical node hosting the sender and the physical node hosting the receiver, respectively. The last column (*testbed*) shows the sum of the costs generated by a link throughout the testbed. The values show that inter-vm links generate almost 10 times more costs than intra-vm links. Inter-pnode links generate two times more costs than inter-vm and about 20 times more costs than intra-vm links. However, in case of inter-pnode links, the costs are distributed to two physical nodes and, therefore, the physical node costs (*pnode*) are slightly lower than the costs generated by inter-vm links.

Figure 5.10 shows the results for a router chain with 10 virtual nodes per segment. The right side shows the measured resource requirements (CPU) for a number of segments varying from 4 to 12. The upper part shows the results for inter-vm links between the segments and the lower part shows the results for inter-pnode links. The left side shows the resource requirements calculated by the testbed model on the basis of the measured data rates of the emulation. We differentiate the resource requirements by the resources running the host-os and the resources allocated by the VMs.

By comparing the calculated resource requirements with the actual resource requirements allocated during the experiment, we can rate the accuracy of the testbed model. As can be seen from the figure, the calculated resource requirements match with the measured resource requirements. In this scenario, we are able to calculate the load with an average error of about 10%.

In order to ensure that the testbed model can calculate the resource requirements for an arbitrary scenario, we evaluated the gathered cost matrix  $\kappa$  in a second application scenario. The scenario, depicted in Figure 5.11, consists of 3,600 video cameras acting

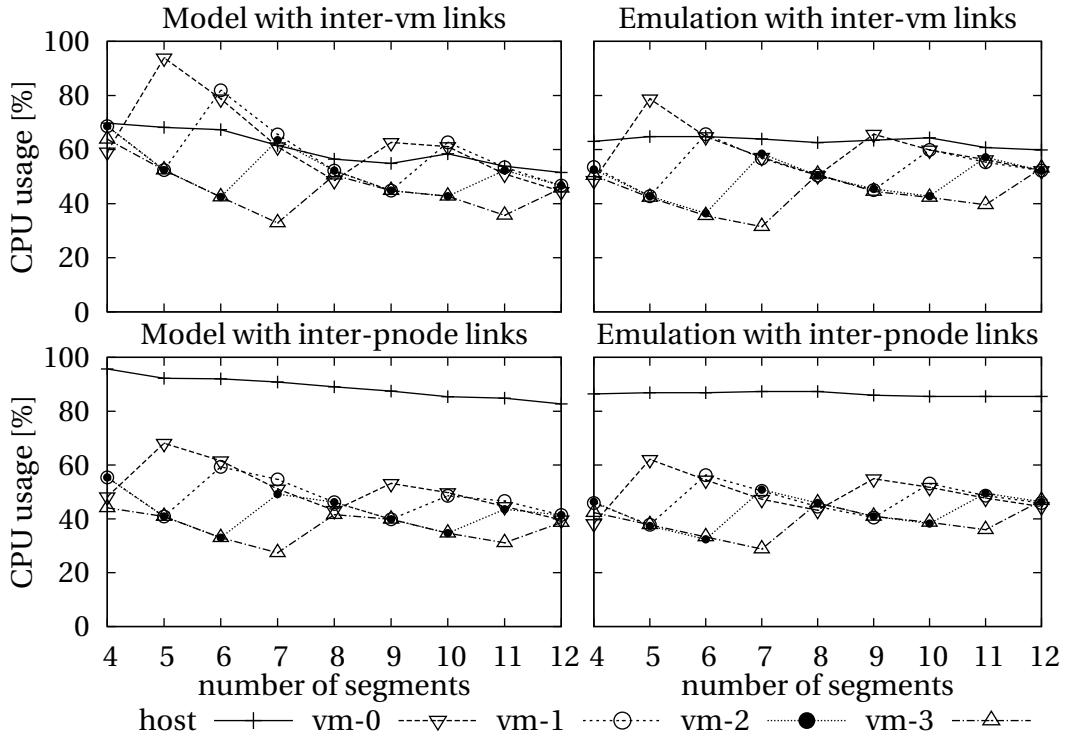


Figure 5.10: Model accuracy for router chain with 10 virtual nodes per segment

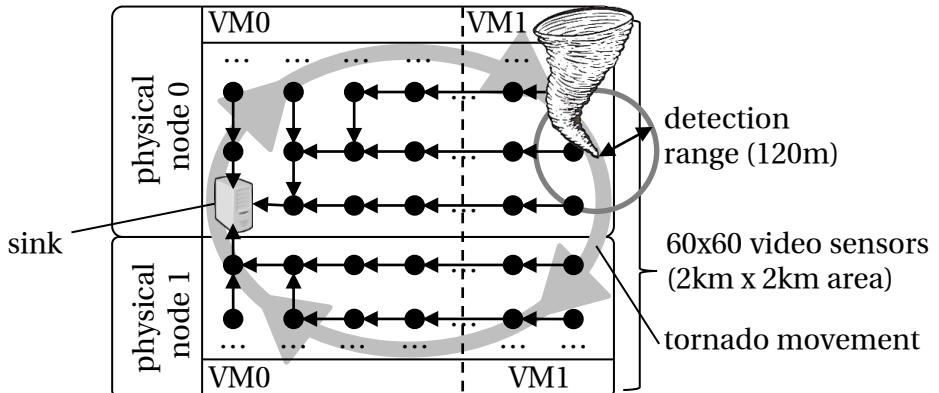


Figure 5.11: Scenario to evaluate the testbed model accuracy

as sensor nodes arranged in a regular grid. The one node on the left acts as a sink. The routes in the scenario are using geometric routing. The nodes are running a synthetic application which monitors a moving virtual tornado. A full circle of the tornado takes 300 s. In case the tornado is in sight (distance between the sensor and the tornado is less than 120 m), the sensor sends a 2 Mbps stream to the sink, otherwise a 16 kbps stream. The stream consists of UDP packets with a size of 1,024 bytes. The nodes are distributed to two physical nodes each running two virtual machines. We restricted the access of

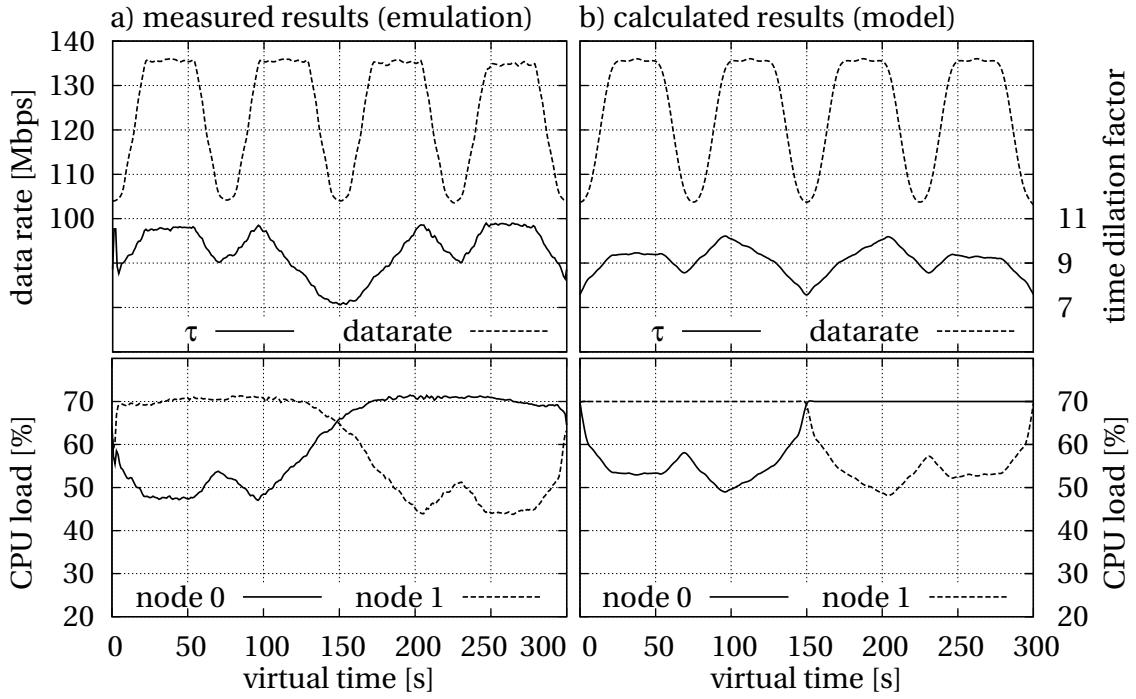


Figure 5.12: Comparison measured and calculated results for testbed model accuracy evaluation

each physical node to only two CPUs of the available eight CPUs. Each virtual machine, equipped with 1.5 GB memory, hosts 900 virtual nodes.

The measured data rate at the sink is illustrated in the upper part Figure 5.12a. The upper part of Figure 5.12b shows for the same setup the calculated data rate based on the scenario description and the testbed model (cf. Section 5.2). Here, we assume a resource consumption of the SuT  $\lambda$  of 70 MCps (million cycles per second) with the tornado in sight and 12 MCps without the tornado in sight, respectively. These values are determined by measuring the resource consumption of the SuT using an experiment with a single instance of the SuT. Comparing the measured data rate with the calculated data rate shows that the emulation results are not biased. A resource congestion during the emulation would result in message loss and, therefore, in a reduced data rate at the sink.

In order to evaluate the accuracy of the testbed model, we measured the time dilation factor<sup>32</sup>  $\tau$  and the load of the maximum loaded CPU during the experiment. The measurements are visualized in Figure 5.12a. As can be seen in the figure, by adapting

<sup>32</sup>To ease readability, Figure 5.12 shows the time dilation factor  $\tau$  as the quotient of the virtual time and the real time using the linear definition of Gupta et al. [GYM<sup>+</sup>06] instead of our logarithmic definition (cf. Section 4.4).

the time dilation factor, the load of the maximum loaded CPU stays at about 70 % which corresponds the used *overload* threshold (cf. Section 4.4). Only in case both physical nodes experience about the same load (at time 150 s), the time dilation factor slightly drops to about 65 % which is still at the upper quarter of the *reasonable load* range (cf. Section 4.4). Additionally to the measurements, we calculated the load of the CPUs and, based on their load, the time dilation factor  $\tau$  using the testbed model. The calculated data are shown in Figure 5.12b.

Comparing the calculated and measured results, we can conclude that our model allows for an accurate load calculation of the system components based on the scenario specification. The calculated resource requirements match with the measured results with an average error of about 8.9 %. The comparison of the calculated experiment runtime (testbed model) of 2,736 s with the actual runtime (emulation) of 2,853 s shows an error of about 4.3%.

Gathering the cost model by an independent experiment ensures that the cost matrix  $\kappa$  was not trained for the experiment. Therefore, we can conclude that the testbed model is able to accurately calculate the resource requirements for an arbitrary experiment.

### 5.5.2 Initial Node Placement

In order to evaluate the experiment runtime of the placement approaches, we use the following nine scenarios. (1) A *Wlan* model with 1,892 nodes randomly distributed over the roads of the inner city of Stuttgart with wireless communication. Due to the road network, the node density varies, resulting in nodes with a high and low number of links. The links are established using a ray tracing-based radio model [Ste08]. (2,3,4) Scenarios *NetworkMap* [MBB00], *Internet*<sup>33</sup> and *AT&T*<sup>34</sup> are based on snapshots of Internet topology gathered by RocketFuel [SMWA04] with 2,376, 2,113 and 753 routers, respectively. (5) A *Grid* model with 1,600 sensor nodes arranged in a regular square grid. Here, direct neighboring nodes can communicate. (6) A *Ring* scenario with 100 nodes arranged in a ring. (7) A *Campus* model with a network of connected campus sites, which is often used to evaluate scalability in the field of parallel simulation [KSU05, ELL09]. We use 20 campuses with a total of 5,480 nodes. (8,9) The final scenarios are generated by the topology generator BRITE [MLMB01]. In the *Waxman* scenario 1,250 nodes are

<sup>33</sup>*PoP-level ISP maps*. Data file policy-dist.tar.gz available at <http://www.cs.washington.edu/research/networking/rocketfuel/>, 2012

<sup>34</sup>*ISP Maps*. Data file rocketfuel\_maps\_cch.tar.gz available at <http://www.cs.washington.edu/research/networking/rocketfuel/>, 2012

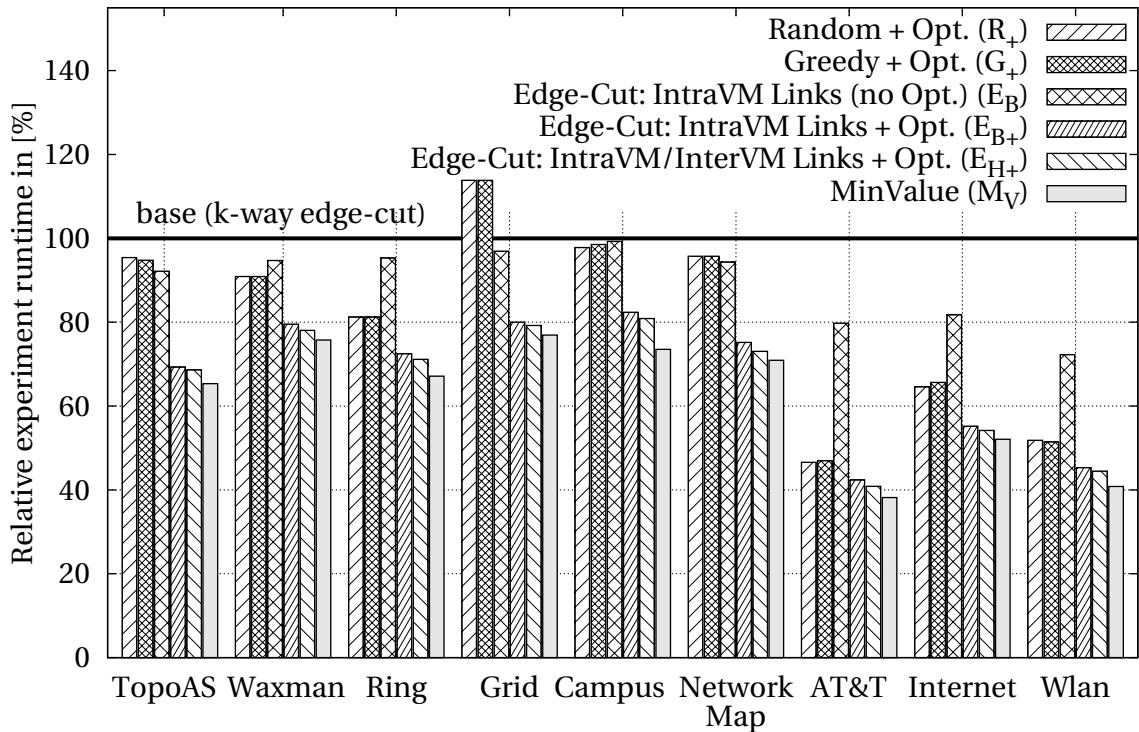


Figure 5.13: Comparison of the experiment runtime using placements calculated by *NETplace* and k-way edge-cut (*metis*)

randomly connected by a Waxman distribution. In *TopoAS* scenario 1,024 nodes are equally distributed to 32 autonomous systems which are connected by a backbone network. In all scenarios, a random link usage between 1 and 100 Mbps is assumed.

The testbeds used in the evaluation contain 2, 4, 8, 16, 32, 64, 128 and 256 CPUs distributed over machines with 1, 2, 4, 8 and 16 CPUs. In total, we are using 34 different testbeds. Since we do not have access to such large testbeds, we used our cost model to calculate the runtime of a placement. For each approach the placement calculation is repeated 200 times. The placements are calculated on Intel Xeon 3 GHz processors.

Figure 5.13 shows the average experiment runtime of the placement strategies relative to a placement calculated by *minimized k-way edge-cut* using the *metis*<sup>35</sup> framework [KK98a]. The figure shows the performance of the greedy approach with the subsequent optimization phase( $G_+$ ), the balanced edge-cut approach without the optimization phase ( $E_B$ ) and with the optimization phase ( $E_{B+}$ ) and the hierarchical edge-cut approach with the optimization phase ( $E_{H+}$ ). Due to the NP-hardness of the placement

<sup>35</sup>As proposed by the authors of *metis* [KK98b], we use *kmetis* to partition a graph into more than eight partitions. Otherwise we use *pmetis*.

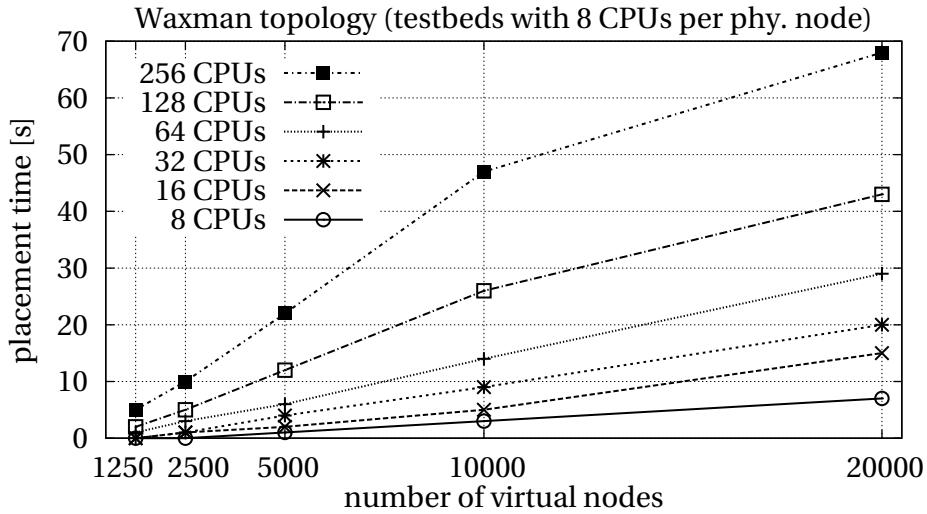
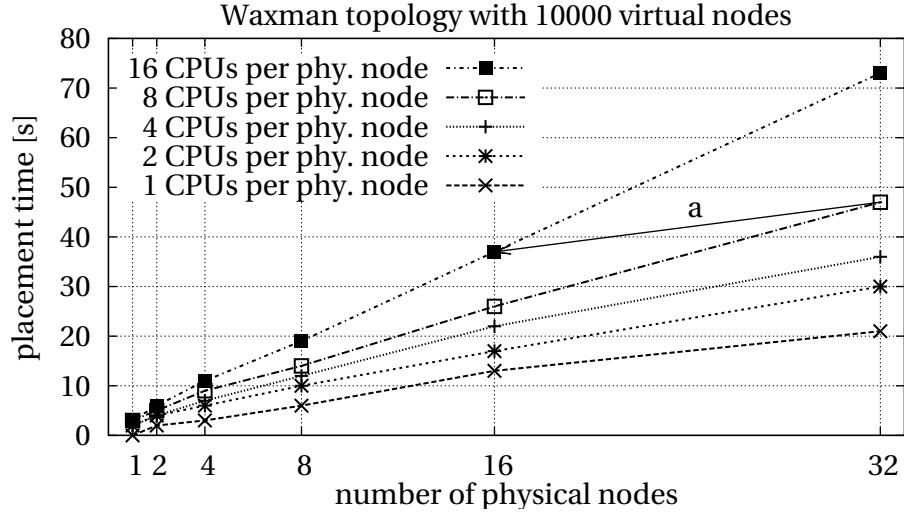
problem we cannot compute the optimum. To estimate the minimum value ( $M_V$ ), we computed for each scenario the best placement out of all runs with all algorithms. Finally, for the purpose of comparison, we included the results for a random placement followed by the placement optimization ( $R_+$ ).

Looking at the performance of  $G_+$ , it turns out that this simple greedy approach can outperform the reference algorithm in 8 of 9 scenarios. Only in the *Grid* scenario the *k-way edge-cut*-based approach results in a better placement. However, the comparison of the randomized approach  $R_+$  with the greedy approach  $G_+$  shows the effectiveness of the placement optimization. After the placement optimization the greedy and the randomized approach have the same experiment runtime. However in case of the greedy approach less optimization steps are required and, therefore, the time to calculate the placement is shorter.

The consideration of the load introduced by intra-vm links ( $E_B$ ) improves the placement in all scenarios compared to the *k-way edge-cut*-based approach. However, in 6 out of 9 scenarios,  $E_B$  results in a worse placement than the greedy approach ( $G_+$ ). The additional optimization phase (see  $E_{B+}$ ) improves the runtime of  $E_B$ . In comparison with the reference algorithm *k-way edge-cut*,  $E_{B+}$  reduces the experiment runtime between 20 % and 50 %. The hierarchical approach  $E_{H+}$  can only slightly further reduce the experiment runtime compared to  $E_{B+}$ . As shown in the figure, the average experiment runtime achieved with  $E_{H+}$  is almost as good as the minimal calculated runtime and compared to the reference algorithm, we can reduce the runtime of network experiments by up to 60 %.

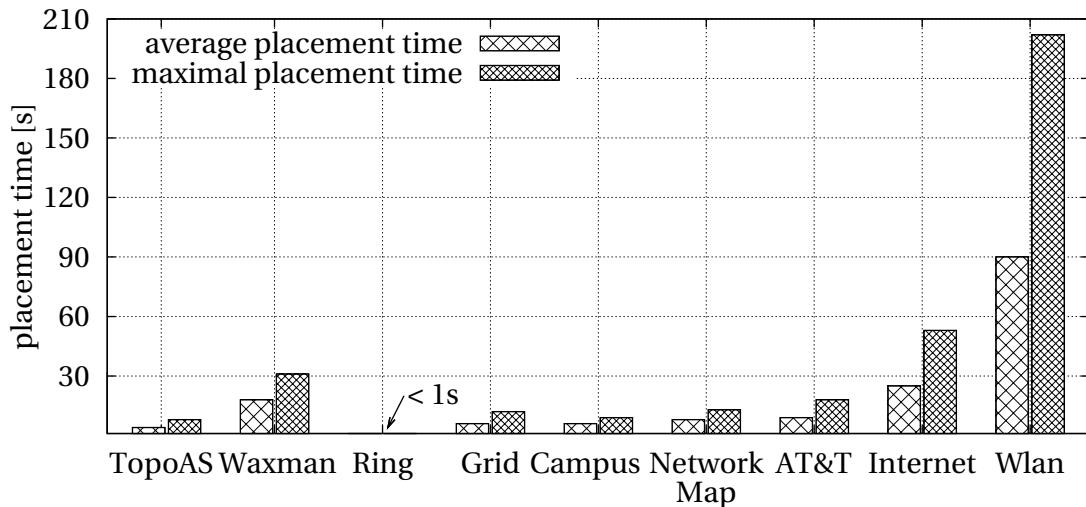
The time to run an experiment consists of the time to calculate the placement and the experiment execution itself. Therefore, the benefit of a faster experiment experiment execution is reduced by the time to calculate the placement using *NETplace*. Therefore, we evaluate the required time to calculate the placement. In the following, we first investigate the impact of the number of virtual nodes on the placement time. Second, we evaluate the placement time for different testbed sizes. Finally, we show the required placement time for the previously introduced nine scenarios.

In order to evaluate the scalability of the placement strategies, we compare the placement runtime of the hierarchical approach  $E_{H+}$  in scenarios with different numbers of virtual nodes. We are using the *Waxman* scenario, because we can easily generate topologies with the same characteristics with respect to node density and link distribution and only vary the number of virtual nodes using the topology generator. For this purpose, we use scenarios with 1,250, 2,500, 5,000, 10,000 and 20,000 nodes. The virtual

Figure 5.14: Scalability of *NETplace*Figure 5.15: Scalability of *NETplace*

networks are placed onto testbeds containing 8 to 256 CPUs with 8 CPUs per physical node.

Figure 5.14 shows the time required to calculate a placement of the virtual nodes onto the testbed. The graph shows that the placement time increases almost linearly with the number of virtual nodes (about 70 seconds for 20,000 virtual nodes). The figure also shows that the time increases sublinearly with the number of CPUs in the testbed. Apart from the figure, there is only a small deviation between the placement runs, which is introduced by the randomized optimization. Even if the placement time varies, the generated placements result in almost the same experiment runtime.


 Figure 5.16: Placement time of the hierarchical approach  $E_{H+}$ 

Instead of varying the number of virtual nodes, we are now varying the testbed size. Here, testbeds with 1 to 32 physical nodes equipped with 1 to 16 CPUs are used. Again, we make use of the generated scenario *Waxman* with 10,000 virtual nodes. Figure 5.15 shows the required placement time for the hierarchical approach  $E_{H+}$ . As can be derived from the figure, the placement time grows linearly with the number of physical nodes. Comparing testbeds with 256 CPUs based on physical nodes with 8 and 16 CPUs per physical node (arrow a in Figure 5.15) shows that increasing the number of CPUs per physical node, while keeping the total number of CPUs the same, reduces the placement time. The same is true for other testbed sizes.

Figure 5.16 shows the average and maximal placement time of the hierarchical approach  $E_{H+}$  for different network topologies. Here, the topology is placed onto a testbed consisting of 32 physical nodes each equipped with 16 CPUs. The figure shows, that the average placement time is about half of the maximal placement time. Except for the *Wlan* scenario, the average placement takes below 30 s and the maximum is below 1 min. The high number of virtual links in the *Wlan* scenario results in a large number of neighboring states in the optimization phase and, therefore, in an increased placement time. However, even for a testbed consisting of 512 CPUs, the placement takes on average about 1.5 min. In comparison with the typical experiment runtime, this overhead is acceptable.

Phase	Action	Time
suspend	suspend all virtual nodes	6ms/vnode
migrate	snapshot virtual node's state	4.6 ms/MB
	state transfer (same physical node)	13.1 ms/MB
	state transfer (different physical node)	15.0 ms/MB
	restore virtual node's state	1.8 ms/MB
change-topology	reattach to virtual topology	200ms/vnode
resume	resume all virtual nodes	3.5 ms/vnode

Table 5.3: Costs for virtual node migration

### 5.5.3 Dynamic Reconfiguration

In order to evaluate *NETbalance*, we have extended our emulation testbed by the dynamic reconfiguration concepts. The basis of our implementation is the extension of OpenVZ's checkpoint/restore functionality [MKK08, Ope12] for capturing frames which are queued in *NETshaper* during the migration of virtual nodes [Bar11]. Additionally, we have extended *NETshaper* to capture statistics of average link data rates, and we have implemented a load monitor for sending the data rates and the CPU usage of virtual nodes to the coordinator. Finally, we have developed a coordinator to calculate the optimized placement and to migrate the virtual nodes.

The evaluation of *NETbalance* is performed in three steps. First, we run a set of micro benchmarks to identify the costs of migration. Second, we emulate a scenario with three virtual nodes and show that the migration does not bias the results. Third, using a synthetic evaluation based on the migration cost model and the testbed model, we evaluate the performance of *NETbalance*. Here, we use the models of the testbed and the migration to calculate the runtime of the experiments with and without *NETbalance*.

The results of the micro benchmarks are summarized in Table 5.3. Here, we measured the costs for creating a snapshot of a virtual node, for transferring the snapshot and for restoring it. Multiplying these costs with the memory footprint of a SuT gives the migration time of a virtual node. Additionally, we measured the time for suspending and resuming an experiment and the time required for modifying the emulated network topology. Both linearly grow with the number of virtual nodes running in a VM. Finally, we measured the size of the state of a virtual node with and without running a minimal SuT which send ICMP ECHO requests<sup>36</sup>. Without the SuT, a snapshot of a virtual node

<sup>36</sup>Internet Control Message Protocol (ICMP) RFC 792, 1981

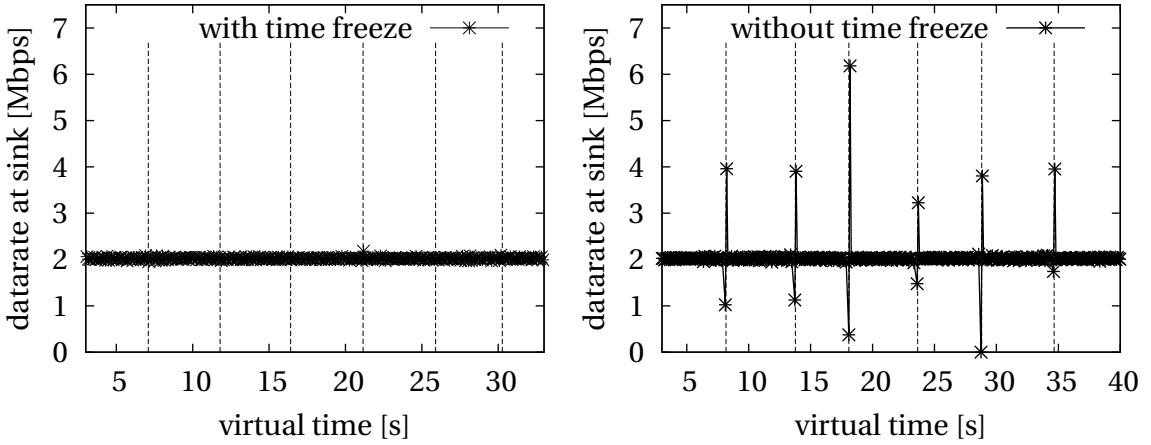


Figure 5.17: Datarate during migration (with and without time freeze)

consumes about 140 kB and with the SuT, the snapshot has a size of 240 kB. However, a more complex SuT requires more memory and, therefore, we evaluate the effect of the virtual node's state size on the performance of *NETbalance* later in this section.

In order to show the emulation accuracy in presence of virtual node migrations, we emulated a scenario with two nodes connected to a third node (*sink*). The two nodes send a stream of UDP packets with 1 Mbps to the third node. At the beginning all nodes are running inside  $VM_1$ . During the experiment all nodes are migrated one by one to  $VM_2$ , and then one by one back to  $VM_1$ .

Figure 5.17 shows the data rate at the sink during the experiment. The right part of the figure shows migration of the virtual nodes without suspending virtual time. Each migration causes a drop followed by a peak in the data rate at the sink. The left part of the figure shows the same scenario, however, we suspended the virtual time during the migration. From the point of the *Software under Test*, the migration takes zero time, and the data rates observed by the sink are not influenced by the migration.

In the following, we show the potential of virtual node migration using four network topologies: *Internet*, *Grid*, *Campus*, and *Waxman*. The topologies are equal to the topologies used for the evaluation of *NETplace* (cf. Section 5.5.2). The evaluated scenario consists of a wired video sensor network with periodic load changes. Each virtual node runs a data source sending a constant data stream of 10 Mbps to a sink. During each experiment, the node acting as the sink changes 15 times which results in large changes of the data flows between the virtual nodes. The routes to reach the sinks are precalculated. We use an initial placement optimized for the data flows to the first sink. If not otherwise stated, the sink changes every 2 min. As a default, we use a testbed with

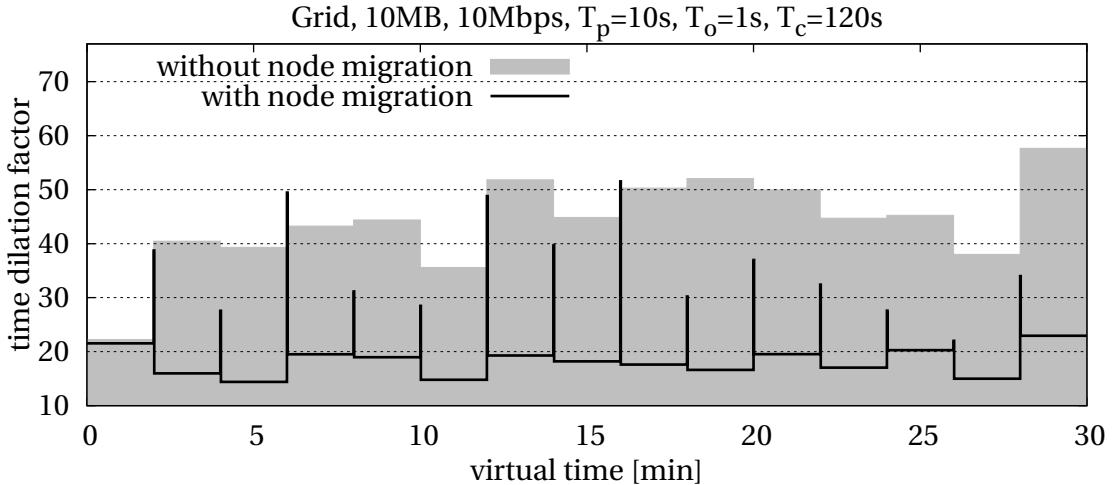


Figure 5.18: Experiment speed over time<sup>37</sup>

8 physical nodes each with 8 CPUs and virtual nodes arranged in the *Grid* topology with a SuT allocating 10 MB memory. The default prediction window  $T_p$  is set to 10 s, and the default optimization time  $T_o$  is 1 s.

We first show the effectiveness of *NETbalance* to calculate an improved placement of virtual nodes. Then, we evaluate the influence of the underlying network topology, the optimization time  $T_o$ , the sink change interval  $T_c$ , the prediction window  $T_p$ , the memory footprint of the SuT, the data rates of the SuT, and the testbed size on the experiment runtime. Each setup is executed 30 times with and without migration of virtual nodes. Our evaluation metric is the relative runtime which is defined as the experiment runtime with *NETbalance* divided by the runtime without using *NETbalance*.

Figure 5.18 shows the required time dilation factor  $\tau$  for running the scenario with the *Grid* topology<sup>37</sup>. The gray area shows  $\tau$  without *NETbalance*. Since the placement is not adapted to the changed sink, it becomes suboptimal after the first sink change which results in an increased  $\tau$ . The black line in the figure shows  $\tau$  in the same scenario using *NETbalance*. As soon as the sink changes,  $\tau$  rises. However, after the deployment of an optimized placement,  $\tau$  decrease again and goes back near to the original level. The different values of  $\tau$  are caused by the location of the sink in the network.

In Figure 5.19, we present the influence of the optimization time  $T_o$ . In contrast to the other evaluations, we change the sink every 10 s which is equal to the prediction window  $T_p$ . Even with a very short time of  $T_o = 0.125$  s (virtual time), *NETbalance* can

<sup>37</sup>To ease readability, Figure 5.18 shows the time dilation factor  $\tau$  as the quotient of the virtual time and the real time using the linear definition of Gupta et al. [GYM<sup>+</sup>06] instead of our logarithmic definition (cf. Section 4.4).

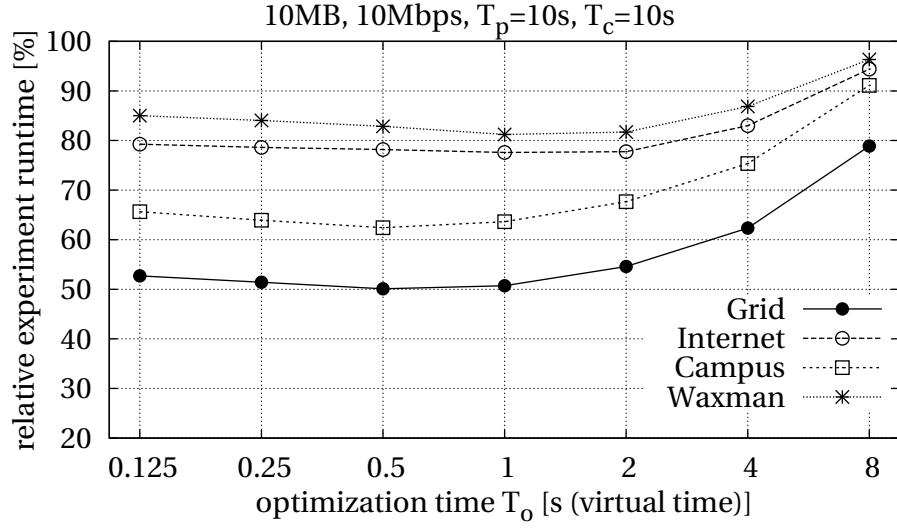


Figure 5.19: Optimization time vs. network topology

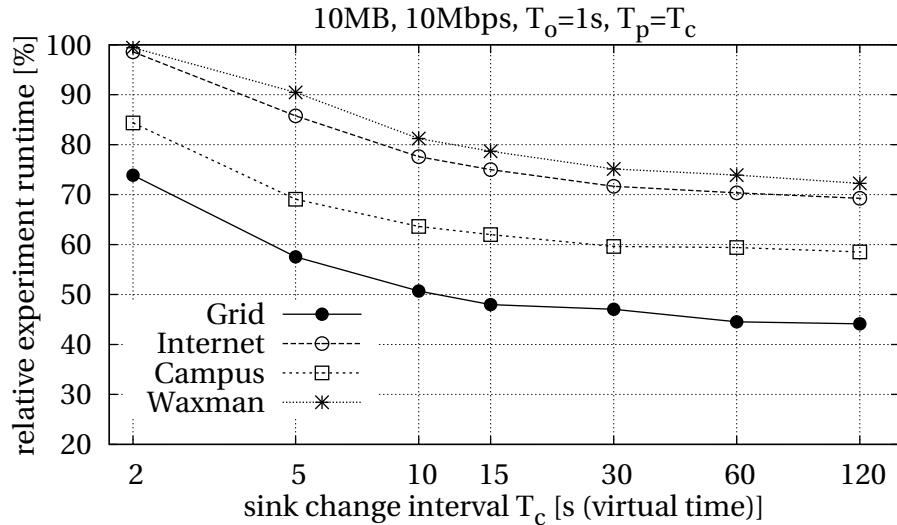


Figure 5.20: Migration benefit vs. sink change interval

reduce the experiment runtime by up to 48 %. Larger optimization times ( $T_o \leq 1$  s) only slightly decrease experiment runtime. The reason is that the increased execution speed is almost compensated by the shorter time  $T_p - T_o$  left for running the experiment with the improved placement. Further increasing  $T_o$  reduces the gain of *NETbalance*, because the short time  $T_p - T_o$  enables only minimal improvements to the virtual node placement. This graph can be generated online during the experiment run, enabling us to learn the optimal value of  $T_o$  for a specific scenario.

Figure 5.20 shows the influence of the sink change interval  $T_c$  on the experiment runtime.

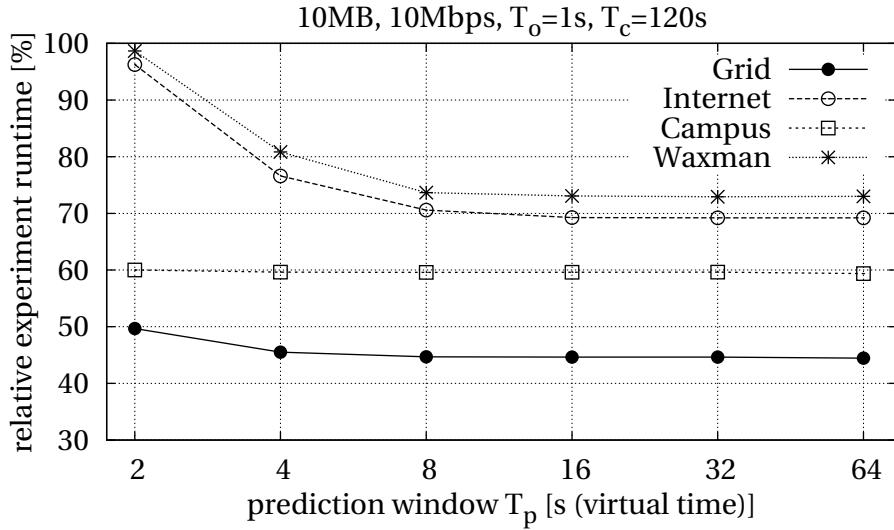


Figure 5.21: Prediction window vs. network topology

Here, we used a prediction window equal to the sink change interval and an optimization time of  $T_o = 1$  s. While increasing the sink change interval, the benefit of the migration increases. The reason is again the increasing time to run the experiment with the optimized placement.

Figure 5.21 shows the experiment runtime for prediction windows  $T_p$  between 1 s and 64 s for the different network topologies. For the *Campus* and the *Grid* scenario small values of  $T_p$  are sufficient for a significant speedup. This mainly comes from the fact that in these topologies small changes in the placement are sufficient to reduce the required  $\tau$  significantly. At the same time, these small changes introduce only small reconfiguration costs which can be compensated even for short  $T_p$ . Regarding all evaluated topologies, a prediction window of 8 s is big enough to improve the placement. A further increase of the prediction time results only in a marginal runtime reduction. The achieved runtime reduction is between 27 % and 55 %.

Since the memory footprint of the SuT mainly determines the reconfiguration costs, we evaluated the required prediction window for different footprint sizes of the SuT. Figure 5.22 shows that increasing the footprint sizes requires larger prediction windows to outweigh the reconfiguration costs and, therefore, to reduce the experiment runtime. As shown in the figure, even for larger SuT with 100 MB of used memory pages, a prediction window of  $T_p \geq 8$  s is sufficient for reducing the experiment runtime.

In contrast to the memory footprint, higher data rates between virtual nodes are beneficial for *NETbalance* (cf. Figure 5.23). Higher data rates result in higher load of the

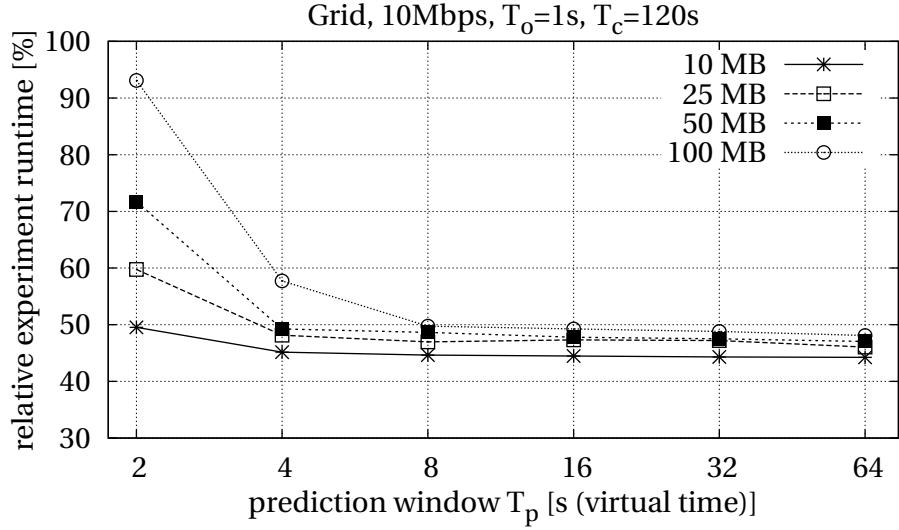


Figure 5.22: Prediction window vs. memory of the SuT

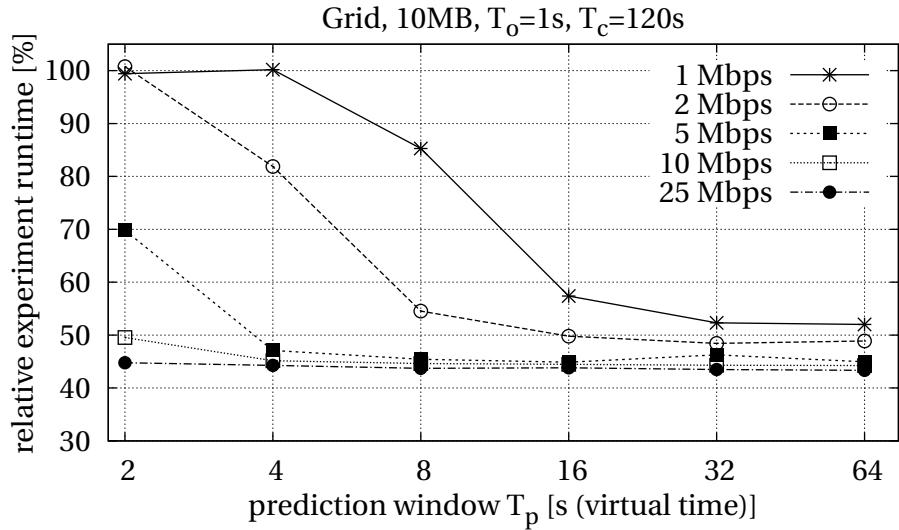


Figure 5.23: Prediction window vs. data rate

physical nodes and, therefore, a higher TDF. The optimization time  $T_o$  and the prediction time  $T_p$  both increase with the TDF. In contrast, the reconfiguration costs  $T_r$  are unaffected. This results in more time for calculating the placement and, also, allows for migrating more virtual nodes because relative to the prediction window the reconfiguration time becomes smaller.

Finally, we evaluate the experiment runtime for different testbed sizes (cf. Figure 5.24). Here, we vary the total number of CPUs from 4 to 512, distributed over physical nodes with 1, 2, 4, and 8 CPUs. For testbeds with up to 64 CPUs, the size of the testbed has only

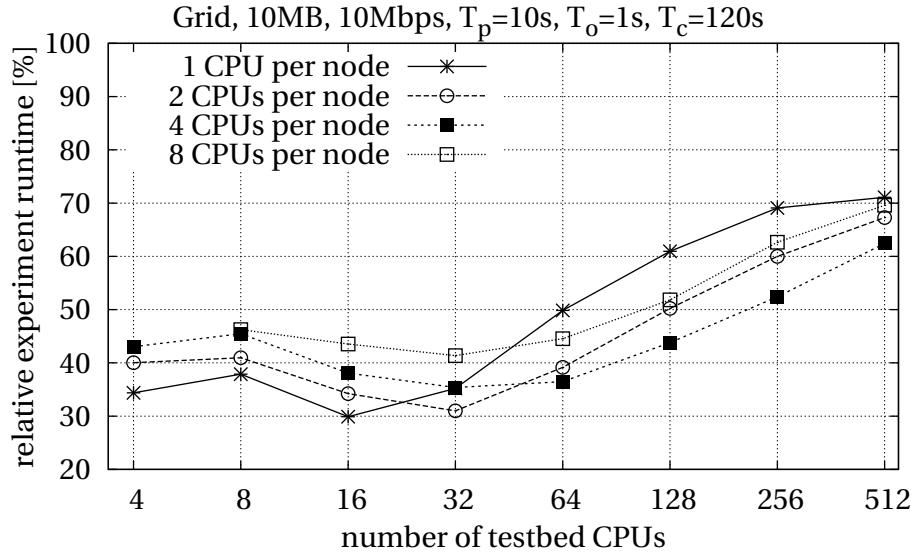


Figure 5.24: Migration benefit vs. testbed size

marginal effects on the relative experiment runtime where a runtime reduction between 50 % and 70 % is achieved. In testbeds with more than 128 CPUs, only few virtual nodes are executed in each VM. Due to this small number, the difference between the unoptimized and the optimized placement becomes small, which limits the performance of *NETbalance*.

Further experiments with other network topologies conform the generality of the presented results. The selected experiments showed, the influence of the SuT's memory footprint, the data rate between virtual nodes, the network topology and the network size. The general effect, that experiments with higher data rates and a smaller memory footprint requires shorter prediction windows is confirmed by all experiments. However, the absolute values vary for different topologies.

## 5.6 Summary

In this chapter, we introduced our approaches to configure a network experiment. The execution of an experiment is defined by the experiment workflow, with its three building blocks: experiment and testbed specification, the initial node placement and the dynamic reconfiguration.

We introduced a testbed model [GHR10] to determine the runtime of an experiment based on the experiment and the testbed configuration. The experiment specification includes the network topology and the SuT. The specification process is supported by two independent software tools: a textual object-oriented scripting language and a graphical user interface. In order to calibrate the testbed model to an actual testbed, we provided an automatic approach based on the execution of a sample scenario.

Based on the testbed model, we introduced *NETplace* [GHR10] to calculate a placement of virtual nodes onto the physical nodes that minimizes the runtime of the experiment. In order to react on changing resource requirements during the experiment execution, we extended our testbed by *NETbalance* [GHR11]. *NETbalance* adapts the placement of virtual nodes by migrating virtual nodes during the experiment execution to reestablish an optimal placement.

Our evaluations showed that our testbed model is able to accurately determine the resource requirements and, thus, the runtime of an experiment based on the experiment specification. The evaluation of multiple network topologies with different testbed sizes showed, that *NETplace* is able to reduce the runtime of an experiment by up to 60 %, in comparison to commonly used initial placement approaches based on *k-way edge-cut*. With respect to the dynamic reconfiguration approach, we showed that virtual nodes can be migrated during the running experiment without biasing the emulation results. Finally, our evaluation demonstrated that *NETbalance* can reduce the experiment runtime by up to 70 %.

# 6 Summary

## 6.1 Conclusions

Software testing is an essential part of the software development. Besides the functional testing, the performance evaluation is an important aspect of the software test. In case of distributed systems, the performance is heavily influenced by the target environment and, therefore, needs to be considered. The methodology of network emulation combines the benefits of network simulation and real work testbeds and constitutes a commonly used approach for performance evaluation of distributed systems. Network emulation enables repeatable network experiments to evaluate unmodified applications and communication protocols in controlled environments. In order to maximize the utility of emulation testbeds, concepts to maximize the supported scenario sizes and to minimize the runtime of the experiments are mandatory.

In this thesis, we focused on mechanisms to increase the scalability of network emulation and on concepts to minimize the runtime of network experiments. Therefore, we first introduced the fundamental concepts of scalable network emulation. Distributed emulation tools such as *NETshaper* are used to reproduce the characteristics of arbitrary networks in an emulation testbed. The number of instances of the *Software under Test* (virtual nodes) is increased by partitioning the physical resources of the emulation testbed using the concept of *node virtualization*. Evaluations showed that virtual nodes based on virtual protocol stacks have an order of magnitude less overhead than virtual machine-based virtual nodes. However, the resource requirements of an emulation experiment running at real time can easily exceed the resources of the physical nodes of a testbed. In order to avoid biased emulation results due to temporary or contin-

uous resource overload, the concept of *virtual time* is applied. Here, the experiment execution is slowed down by a factor. This slowdown results in a reduced resource consumption of the experiment by the same factor. Current systems make use of virtual machines to provide *virtual time* transparently to the *Software under Test*. Besides the overhead of the virtual machine-based node virtualization, these network emulators only support constant execution speed of the experiment. In case of varying resource requirements during the experiment, resources are used suboptimal and, thus, the experiment runtime is unnecessarily extended.

In order to minimize the runtime of network experiments, we developed an architecture to efficiently support node and time virtualization. The basis of our architecture is the combination of node virtualization based on virtual protocol stacks with transparent time virtualization based on virtual machines. We execute one virtual machine per CPU of the physical node and partition the resources of the VM using virtual protocol stacks. While supporting testbeds based on multi-core CPUs, this approach minimizes the communication and memory overhead. The presented architecture is able to make use of virtualization-aware network adapters to improve the access to the emulation network. The concept of trigger-based timers is used to improve the accuracy of the emulation tool. In case of the emulation of high speed networks, our fine-grained timer reduces bursty frame transmissions. In order to overcome the limitations of the commonly used *VLAN*-based link multiplexing scheme, we have developed an approach to increase the number of virtual links in the emulated network topology by reusing *VLANs* for multiple virtual links. Our concept of *adaptive virtual time* allows for minimizing the experiment runtime. Here, we monitor the load of the physical nodes. Based on load reports a controller running on a central coordinator allows for adapting the executing speed of the experiment. The executing speed is adapted to the value that maximizes the resource utilization without overloading the resources.

Besides the efficiency of the emulation architecture, the placement of the virtual nodes onto the physical nodes strongly influences the runtime of an experiment. In order to minimize the runtime, we developed a cost model to determine the resource requirements of a network experiment based on the experiment specification. To support the specification process, we have developed a textual and a graphical user interface. Based on the testbed cost model and the experiment specification, our placement tool *NETplace* is used to calculate an initial placement of the virtual nodes that minimizes the runtime of the experiment. In order to react on resource changes during the experiment, we have developed an extended placement approach called *NETbalance*. Here, varying resource requirements of the *Software under Test* result in a recalculation of the

placement. In case the optimized placement results in a reduced experiment runtime, the new placement is established by migrating virtual nodes. In order to efficiently perform reconfigurations, we have extended our architecture by a network-based file access for the virtual nodes.

Our evaluations showed, that our system has very low memory overhead of only about 300 kB. We showed that, using time virtualization, the performance of our emulation system linearly scales with the execution speed of the experiment. This enables the emulation of scenarios with thousands of virtual nodes per physical node. Additionally, links between virtual nodes can have bandwidths, which are magnitudes larger than the physical network bandwidth. The evaluation of the *adaptive virtual time* demonstrates, that the available CPU and network capacity no longer limits the possible scenario sizes. The scenario size is only limited by the available memory (RAM) and the permissible execution time of an experiment. This is an important step forward in terms of emulation scalability.

The evaluation demonstrated the accuracy of our cost model to determine the resource requirements of an experiment. In terms of experiment runtime minimization, the evaluation results show that our placement strategy *NETplace* can reduce experiment runtime up to 60 % compared to current placement approaches. Evaluation of *NETbalance* showed, that virtual nodes can be migrated during a running emulation experiment without biasing the results. Moreover, the reconfiguration of the virtual nodes' placement allows for reducing the runtime of network experiments by up to 70 % for various network topologies and load characteristics.

Overall, the concepts presented in this thesis constitute a major improvement of the network emulation technology. The efficiency and scalability of our approaches allow for large scale network experiments of unmodified applications and communication protocols. The concepts *NETplace* and *NETbalance* enable scientists to run more experiments in less time, achieving statistically more relevant results. Moreover with *NETbalance* the effort of preparing experiments is drastically decreased as prior knowledge about application behavior is no longer needed.

## 6.2 Promising Research Directions

A recent trend in computer industry is shifting desktop applications running on a common operating system (e.g. Linux) to mobile devices. These devices run customized software stacks (e.g., Android or iOS) including custom libraries and operating systems. Similar trends are present in the area of cloud computing, where cloud operators provide the customers a specialized application interface to access the cloud resources (e.g., Google Cloud). These custom software stacks are typically based on virtual machines which prohibits the execution of these applications on other operating systems. Therefore, a direct evaluation of these applications in an network emulation system is not possible. Generally, nested virtualization [BYDD<sup>+</sup>10] could be used to deploy these VMs to the virtual machines of the network emulator. However, this approach results in a high overhead. A promising research direction is to investigate emulation architectures that efficiently support the evaluation of applications written for mobile operating systems or cloud infrastructures. Initial work on an extension of our network emulator to support the mobile operating system Android has been done in a supervised study thesis by Schirmer [Sch10].

A common approach during the evaluation of distributed systems is to investigate the behavior of an algorithm in presence of environment changes during the runtime of the application. In case of a fault tolerant communication protocol, the reaction to link failures, message loss, network congestion or increased network delay is of interest. Such experiments can be easily conducted by the network emulation approach. However, in complex scenarios such experiments have typically an initialization phase to reach a specific application state of the *Software under Test* (SuT). After reaching the state of interest, the emulated environment is altered and the behavior of the SuT is monitored. Such experiments are typically repeated for a large set of different environment changes. Eliminating the initialization phase would result in a significant reduction of the experiment runtime. Initial work has been done by Burtsev et al. [BRHL09], who have extended the Emulab testbed to create snapshots of a running experiment. Setting up an experiment from such a snapshot can significantly speed up experiment runtime of evaluations that require long initial setup phases.

# List of Figures

2.1	Protocol stack extended by emulation tools . . . . .	32
2.2	Network emulation using simulation frameworks . . . . .	33
2.3	Resource virtualization approaches . . . . .	34
2.4	Virtual nodes based on protocol stack virtualization . . . . .	37
3.1	Physical architecture of the <i>Network Emulation Testbed</i> . . . . .	46
3.2	Components of the <i>Network Emulation Testbed</i> . . . . .	47
4.1	Model of the <i>Software under Test</i> and the virtual hardware . . . . .	52
4.2	AMD server processors . . . . .	55
4.3	Hybrid emulation architecture: node and time virtualization . . . . .	56
4.4	Intra virtual machine communication . . . . .	57
4.5	Alternatives for network access . . . . .	58
4.6	Inter virtual machine communication . . . . .	59
4.7	Emulation with interrupt triggered timers . . . . .	60
4.8	Emulation with event triggered timers . . . . .	61
4.9	Example network topology . . . . .	62
4.10	VLAN-based emulation . . . . .	63
4.11	VLAN-based emulation in presence of node virtualization . . . . .	63
4.12	Scalable VLAN-based emulation architecture . . . . .	66
4.13	Feedback loop for TDF adaptation . . . . .	72
4.14	Thresholds used for TDF adaptation . . . . .	75
4.15	Accuracy of bandwidth emulation ( <i>netperf</i> in UDP mode) . . . . .	80
4.16	Accuracy of delay emulation (ping) . . . . .	80
4.17	Experiment execution speed vs. scenario size . . . . .	81
4.18	Memory consumption of virtual nodes . . . . .	82
4.19	Evaluation scenario for TDF adaptation . . . . .	83
4.20	Load-based TDF adaptation . . . . .	84
4.21	Effectiveness of the TDF adaptation . . . . .	85

4.22 Number of control messages sent by virtual routers running OLSR . . . . .	86
4.23 Experiment runtime of OLSR scenario . . . . .	87
4.24 Number of load reports . . . . .	88
4.25 Number of TDF switches . . . . .	88
4.26 Clock drift without synchronization protocol . . . . .	89
4.27 Clock drift with synchronization protocol . . . . .	89
4.28 Distribution of clock drift . . . . .	90
5.1 Experiment workflow using initial placement and dynamic reconfiguration	93
5.2 Sample network topology . . . . .	96
5.3 NETcaptain . . . . .	98
5.4 Link types . . . . .	100
5.5 Runtime optimal partitioning . . . . .	104
5.6 Transparent migration of virtual nodes . . . . .	112
5.7 State of a virtual node . . . . .	113
5.8 Server-based file system of virtual nodes . . . . .	113
5.9 Scenario to determine testbed's cost matrix . . . . .	118
5.10 Model accuracy for router chain . . . . .	120
5.11 Scenario to evaluate the testbed model accuracy . . . . .	120
5.12 Measured vs. calculated results . . . . .	121
5.13 Performance of the placement approaches . . . . .	123
5.14 Scalability of <i>NETplace</i> . . . . .	125
5.15 Scalability of <i>NETplace</i> . . . . .	125
5.16 Placement time of the hierarchical approach $E_{H+}$ . . . . .	126
5.17 Datarate during migration . . . . .	128
5.18 Experiment speed over time $3637^{36}$ . . . . .	129
5.19 Optimization time vs. network topology . . . . .	130
5.20 Migration benefit vs. sink change interval . . . . .	130
5.21 Prediction window vs. network topology . . . . .	131
5.22 Prediction window vs. memory of the SuT . . . . .	132
5.23 Prediction window vs. data rate . . . . .	132
5.24 Migration benefit vs. testbed size . . . . .	133

# List of Tables

4.1	Properties of common Ethernet switches . . . . .	64
4.2	Table mapping MAC addresses to virtual links and virtual machines . . .	65
4.3	Granularity comparison for linear and logarithmic time dilation factors .	72
5.1	Emulation costs for different virtual link types . . . . .	101
5.2	Cost matrix $\kappa$ of our emulation testbed . . . . .	119
5.3	Costs for virtual node migration . . . . .	127

# List of Algorithms

1	TDF adaptation process . . . . .	76
2	Text-based scenario description using a Ruby script . . . . .	97

# Publications

- [GHR09a] Andreas Grau, Klaus Herrmann, and Kurt Rothermel. Efficient and Scalable Network Emulation Using Adaptive Virtual Time. In *ICCCN'09: Proceedings of the 18th International Conference on Computer Communications and Networks*, pages 1–6, San Fransisco, CA, USA, August 3–6, 2009. IEEE.
- [GHR09b] Andreas Grau, Klaus Herrmann, and Kurt Rothermel. Exploiting Emulation Testbeds for Security Experiments. In *Proceedings of the Workshop on Experimental platforms for Internet resilience, security and stability research (invited paper)*, pages 1–2, Brussels, Belgium, June 19, 2009. JRC European Commission.
- [GHR10] Andreas Grau, Klaus Herrmann, and Kurt Rothermel. NETplace: Efficient Runtime Minimization of Network Emulation Experiments. In *SPECTS'10: Proceedings of the International Symposium on Performance Evaluation of Computer and Telecommunication Systems*, pages 265–272, Ottawa, ON, Canada, July 11–14, 2010. IEEE Communications Society.
- [GHR11] Andreas Grau, Klaus Herrmann, and Kurt Rothermel. NETbalance: Reducing the Runtime of Network Emulation using Live Migration. In *ICCCN'11: Proceedings of the 20th International Conference on Computer Communications and Networks*, pages 1 – 6, Maui, HI, USA, July 31 – August 4, 2011.
- [GHR12] Andreas Grau, Klaus Herrmann, and Kurt Rothermel. Scalable Network Emulation - The NET Approach. *Journal of Communications - Special Issue: Advances in Communications and Networking - II*, 7(1):3–16, January 2012.
- [GMHR08] Andreas Grau, Steffen Maier, Klaus Herrmann, and Kurt Rothermel. Time Jails: A Hybrid Approach to Scalable Network Emulation. In *PADS'08: Proceedings of the 22nd ACM/IEEE/SCS Workshop on Principles of Advanced and Distributed Simulation*, pages 1–8, Rome, Italy, June 3–6, 2008. IEEE Computer Society.

- [GWS06] Andreas Grau, Harald Weinschrott, and Christopher Schwarzer. Evaluating the Scalability of Virtual Machines for Use in Computer Network Emulation. Case Study, Universität Stuttgart, Institute for Parallel and Distributed Systems, Distributed Systems, Stuttgart, Germany, October 2006.
- [MGWR07] Steffen Maier, Andreas Grau, Harald Weinschrott, and Kurt Rothermel. Scalable Network Emulation: A Comparison of Virtual Routing and Virtual Machines. In *ISCC'07: Proceedings of the IEEE Symposium on Computers and Communications*, pages 395–402, Aveiro, Portugal, July 1–4, 2007. IEEE Computer Society.

# Supervised Student Theses

- [Bar11] Sebastian Bartmann. Migration virtueller Knoten in einer zeitvirtualisierten Emulationsumgebung. Diploma thesis, no. 3101, Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Stuttgart, Germany, January 26, 2011.
- [Ego08] Alexander Egorenkov. Protocol for Epoch Switching in a Distributed Time Virtualized Emulation Environment. Diploma thesis, no. 2749, Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Stuttgart, Germany, September 2, 2008.
- [Pak08] Frederik Pakai. Adaptation of the Time Dilation Factor in a Time Virtualized Emulation Environment. Diploma thesis, no. 2795, Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Stuttgart, Germany, December 31, 2008.
- [Sch10] Markus Schirmer. Distributed Emulation of Mobile Applications in TVEE. Study thesis, no. 2273, Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Stuttgart, Germany, November 3, 2010.
- [Sch11a] Markus Schirmer. Scalable Emulation of Network Links between Virtual Nodes. Diploma thesis, no. 3132, Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Stuttgart, Germany, June 30, 2011.
- [Sch11b] Frank Schuh. Entwicklung eines skalierbaren Emulationsrahmenwerks für mobile Knoten. Diploma thesis, no. 3170, Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Stuttgart, Germany, October 11, 2011.
- [Zho11] Kai Zhou. Development of a Load Model for Distributed Systems. Diploma thesis, no. 3082, Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Stuttgart, Germany, March 23, 2011.

# Bibliography

[ABKM01] David Andersen, Hari Balakrishnan, Frans Kaashoek, and Robert Morris. Resilient overlay networks. In *SOSP'01: Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pages 131–145, Banff, Alberta, Canada, October 21–24 2001. ACM. (Cited on pages 13, 25, 41, and 42)

[AC06] George Apostolopoulos and Constantinos Chasapis. V-eM: A Cluster of Virtual Machines for Robust, Detailed, and High-Performance Network Emulation. Technical Report 371, ICS-FORTH, Greece, January 2006. (Cited on pages 13, 25, 26, 35, 37, 43, 64, 103, and 105)

[AD99] Mohit Aron and Peter Druschel. Soft Timers: Efficient Microsecond Software Timer Support for Network Processing. In *SOSP'99: Proceedings of the 17th ACM Symposium on Operating Systems Principles*, pages 232–246, Charleston, SC, USA, December 12–15, 1999. (Cited on page 60)

[AK10] Emmanuel Arzuaga and David R. Kaeli. Quantifying Load Imbalance on Virtualized Enterprise Servers. In *WOSP/SIPEW'10: Proceedings of the First Joint WOSP/SIPEW International Conference on Performance Engineering*, pages 235–242, San Jose, CA, USA, January 28–30, 2010. ACM. (Cited on page 110)

[And02] David G. Andersen. Theoretical Approaches to Node Assignment. Unpublished Manuscript, December 2002. (Cited on page 103)

[AOC<sup>+</sup>10] Alberto Alvarez, Rafael Orea, Sergio Cabrero, Xabiel G. Pañeda, Roberto García, and David Melendi. Limitations of Network Emulation with Single-Machine and Distributed ns-3. In *SIMUTools'10: Proceedings of the 3rd International ICST Conference on Simulation Tools and Techniques*, pages 67:1–67:9, Torremolinos, Malaga, Spain, March 15–19, 2010. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering). (Cited on pages 17, 33, and 42)

---

[Ari12] Arista 7148S Datasheet. <http://www.aristanetworks.com/media/system/pdf/AristaProductQuickReferenceGuide.pdf>, 2012. (Cited on page 64)

[ASR<sup>+</sup>10] Sherif Akoush, Ripduman Sohan, Andrew Rice, Andrew W. Moore, and Andy Hopper. Predicting the Performance of Virtual Machine Migration. In *MASCOTS'10: Proceedings of the International Symposium on Modeling, Analysis, and Simulation of Computer Systems*, volume 0, pages 37–46, Los Alamitos, CA, USA, August 17–19, 2010. IEEE Computer Society. (Cited on page 110)

[Bar11] Sebastian Bartmann. Migration virtueller Knoten in einer zeitvirtualisierten Emulationsumgebung. Diploma thesis, no. 3101, Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Stuttgart, Germany, January 26, 2011. (Cited on pages 94, 116, and 127)

[BD02] Tom Boyd and Partha Dasgupta. Process Migration: A Generalized Approach Using a Virtualizing Operating System. In *ICDCS'02: Proceedings of the 22nd International Conference on Distributed Computing Systems*, pages 385–392, Vienna, Austria, July 2–5, 2002. IEEE Computer Society. (Cited on page 110)

[BDF<sup>+</sup>03] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the Art of Virtualization. In *SOSP'03: Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pages 164–177, Bolton Landing, NY, USA, October 19–22, 2003. ACM Press. (Cited on pages 35, 36, 39, 43, 54, 58, 73, 78, 82, and 83)

[BDM99] Gaurav Banga, Peter Druschel, and Jeffrey C. Mogul. Resource Containers: A New Facility for Resource Management in Server Systems. In *OSDI'99: Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, pages 45–58, New Orleans, LA, USA, February 22–25, 1999. USENIX Association. (Cited on pages 16 and 37)

[Bel05] Fabrice Bellard. QEMU, a Fast and Portable Dynamic Translator. In *ATC'05: Proceedings of the USENIX Annual Technical Conference*, pages 41–46, Anaheim, CA, USA, April 10–15, 2005. USENIX Association. (Cited on pages 34 and 35)

[BF11] Nikos Baltas and Tony Field. Software Performance Prediction with a Time Scaling Scheduling Profiler. In *MASCOTS'11: Proceedings of the 19th*

*International Symposium on Modeling, Analysis Simulation of Computer and Telecommunication Systems*, pages 107–116, Singapore, July 25–27, 2011. IEEE. (Cited on pages 17 and 39)

[BFH<sup>+</sup>06] Andy Bavier, Nick Feamster, Mark Huang, Larry Peterson, and Jennifer Rexford. In VINI Veritas: Realistic and Controlled Network Experimentation. In *SIGCOMM'06: Proceedings of the ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, Pisa, Italy, September 11–15, 2006. (Cited on page 41)

[BL03] Paul Barford and Larry Landweber. Bench-style Network Research in an Internet Instance Laboratory. *ACM SIGCOMM Computer Communication Review*, 33:21–26, July 2003. (Cited on pages 16, 26, and 31)

[Boc12] Bochs IA-32 Emulator Project. <http://bochs.sourceforge.net>, 2012. (Cited on pages 34 and 35)

[BQ06a] Franck Cappello Benjamin Quetier, Vincent Neri. Scalability Comparison of Four Host Virtualization Tools. *Journal of Grid Computing*, 5(1):83–98, 2006. (Cited on pages 16 and 36)

[BQ06b] Franck Cappello Benjamin Quetier, Vincent Neri. Selecting A Virtualization System For Grid/P2P Large Scale Emulation. In *EXPGRID'06: Proceedings of the Workshop on Experimental Grid Testbeds for the Assessment of Large-scale Distributed Applications and Tools*, Paris, France, June 19–23, 2006. (Cited on page 37)

[BRHL09] Anton Burtsev, Prashanth Radhakrishnan, Mike Hibler, and Jay Lepreau. Transparent Checkpoints of Closed Distributed Systems in Emulab. In *EuroSys'09: Proceedings of the 4th ACM European Conference on Computer Systems*, pages 173–186, Nürnberg, Germany, April 1–3, 2009. ACM. (Cited on pages 112 and 138)

[BVB06] Craig Bergstrom, Srinidhi Varadarajan, and Godmar Back. The Distributed Open Network Emulator: Using Relativistic Time for Distributed Scalable Simulation. In *PADS'06: Proceedings of the 20th Workshop on Principles of Advanced and Distributed Simulation*, pages 19–28, Singapore, May 23–26, 2006. IEEE Computer Society. (Cited on pages 40, 42, and 70)

[BYDD<sup>+</sup>10] Muli Ben-Yehuda, Michael D. Day, Zvi Dubitzky, Michael Factor, Nadav Har'El, Abel Gordon, Anthony Liguori, Orit Wasserman, and Ben-Ami Yas-sour. The Turtles Project: Design and Implementation of Nested Virtualiza-

---

tion. In *OSDI'10: Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, pages 1–6, Vancouver, BC, Canada, October 4–6, 2010. USENIX Association. (Cited on page 138)

[BYMX<sup>+</sup>06] Muli Ben-Yehuda, Jon Mason, Jimi Xenidis, Orran Krieger, Leendert van Doorn, Jun Nakajima, Asit Mallick, and Elsie Wahlig. Utilizing IOMMUs for Virtualization in Linux and Xen. In *OLS'06: Proceedings of the Ottawa Linux Symposium*, pages 71–86, Ottawa, ON, Canada, July 19–22, 2006. (Cited on page 36)

[CB10b] N.M. Mosharaf Kabir Chowdhury and Raouf Boutaba. A Survey of Network Virtualization. *Computer Networks.*, 54:862–876, April 2010. (Cited on page 41)

[CBMP04] Jeffrey Considine, John W. Byers, and Ketan Meyer-Patel. A Constraint Satisfaction Approach to Testbed Embedding Services. *ACM SIGCOMM Computer Communication Review*, 34(1):137–142, 2004. (Cited on pages 14, 27, 103, and 105)

[CCR<sup>+</sup>03] Brent Chun, David Culler, Timothy Roscoe, Andy Bavier, Larry Peterson, Mike Wawrzoniak, and Mic Bowman. PlanetLab: An Overlay Testbed for Broad-Coverage Services. *ACM SIGCOMM Computer Communication Review*, 33(3):00–00, July 2003. (Cited on pages 13, 16, 25, 26, 31, 41, 42, 50, and 103)

[CFDL08] Carlo Caini, Rosario Firrincieli, Renzo Davoli, and Daniele Lacamera. Virtual integrated TCP testbed (VITT). In *TridentCom'08: Proceedings of the 4th International Conference on Testbeds and Research Infrastructures for the Development of Networks & Communities*, pages 36:1–36:6, Innsbruck, Austria, March 18–20, 2008. ICST. (Cited on page 43)

[CFH<sup>+</sup>80] M. D. Canon, D. H. Fritz, J. H. Howard, T. D. Howell, M. F. Mitoma, and J. Rodriguez-Rosell. A Virtual Machine Emulator for Performance Evaluation. *Communications of the ACM*, 23(2):71–80, 1980. (Cited on pages 14, 17, 26, 38, 42, 43, 69, and 70)

[CFH<sup>+</sup>05] Christopher Clark, Keir Fraser, Steven Hand, Jacob Gorm Hansen, Eric Jul, Christian Limpach, Ian Pratt, and Andrew Warfield. Live Migration of Virtual Machines. In *NSDI'05: Proceedings of the 2nd Symposium on Networked Systems Design and Implementation*, pages 273–286, Berkeley, CA, USA, May 2–4, 2005. USENIX Association. (Cited on pages 110, 111, and 112)

[CGMV07] Roberto Canonico, Pasquale Di Gennaro, Vittorio Manetti, and Giorgio Ventre. Virtualization Techniques in Network Emulation Systems. In *Euro-*

*Par'07: Proceedings of the Conference on Parallel Processing*, pages 144–153, Rennes, France, August 28–31, 2007. (Cited on pages 16 and 36)

[CH07] Shefali Chinni and Radhakrishna Hiremane. Virtual Machine Device Queues. *White Paper: Intel Virtualization Technology*, 2007. (Cited on page 59)

[Cis12] Cisco ME 4924-10G Datasheet. [http://www.cisco.com/en/US/prod/collateral/switches/ps6568/ps7009/product\\_data\\_sheet0900aecd8052f36b.pdf](http://www.cisco.com/en/US/prod/collateral/switches/ps6568/ps7009/product_data_sheet0900aecd8052f36b.pdf), 2012. (Cited on page 64)

[CM79] K. Mani Chandy and Jayadev Misra. Distributed Simulation: A Case Study in Design and Verification of Distributed Programs. *IEEE Transactions on Software Engineering*, SE-5(5):440 – 452, September 1979. (Cited on page 40)

[CR10] Marta Carbone and Luigi Rizzo. Dummynet Revisite. *ACM SIGCOMM Computer Communication Review*, 40(2):12–20, 2010. (Cited on pages 16 and 32)

[Cre81] R. J. Creasy. The Origin of the VM/370 Time-sharing System. *IBM Journal of Research and Development*, 25:483–490, September 1981. (Cited on pages 16 and 34)

[CS03] Mark Carson and Darrin Santay. NIST Net - A Linux-based Network Emulation Tool. *ACM SIGCOMM Computer Communication Review*, 33(3):111–126, 2003. (Cited on pages 16 and 32)

[CWdO<sup>+</sup>06] Jедидиа R. Crandall, Gary Wassermann, Daniela A. S. de Oliveira, Zhen-dong Su, S. Felix Wu, and Frederic T. Chong. Temporal search: detecting hidden malware timebombs with virtual machines. In *ASPLOS'06: Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 25–36, San Jose, CA, USA, October 21–25, 2006. ACM. (Cited on pages 17 and 38)

[D-l12] D-link DGS-6600 Datasheet. <ftp://ftp.dlink.it/Datasheets/DGS-6604.pdf>, 2012. (Cited on page 64)

[DGCV09] Pasquale Di Gennaro, Roberto Canonico, and Giorgio Ventre. NEPTUNE: Network Emulation for Protocol Tuning and Evaluation. In *SIMUTools'09: Proceedings of the 2nd International Conference on Simulation Tools and Techniques*, pages 36:1–36:2, Rome, Italy, March 2–6, 2009. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering). (Cited on page 43)

---

[DHB<sup>+</sup>08] Marcel Dischinger, Andreas Haeberlen, Ivan Beschastnikh, Krishna P. Gummadi, and Stefan Saroiu. SatelliteLab: Adding Heterogeneity to Planetary-Scale Network Testbeds. *ACM SIGCOMM Computer Communication Review*, 38:315–326, August 2008. (Cited on page 42)

[Dik01] Jeff Dike. A user-mode port of the Linux kernel. In *ALS’01: Proceedings of the 5th Annual Linux Showcase and Conference*, Oakland, CA, USA, November 5–10, 2001. (Cited on page 35)

[DO91] Fred Douglos and John Ousterhout. Transparent Process Migration: Design Alternatives and the Sprite Implementation. *Software: Practice and Experience*, 21:757–785, July 1991. (Cited on page 110)

[DO00] Peter A. Dinda and David R. O’Hallaron. Host load prediction using linear models. *Cluster Computing*, 3:265–280, 2000. (Cited on page 115)

[EGH<sup>+</sup>10] Norbert Egi, Adam Greenhalgh, Mark Handley, Mickael Hoerdt, Felipe Huici, Laurent Mathy, and Panagiotis Papadimitriou. A Platform for High Performance and Flexible Virtual Routers on Commodity Hardware. *ACM SIGCOMM Computer Communication Review*, 40(1):127–128, 2010. (Cited on page 41)

[Ego08] Alexander Egorenkov. Protocol for Epoch Switching in a Distributed Time Virtualized Emulation Environment. Diploma thesis, no. 2749, Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Stuttgart, Germany, September 2, 2008. (Cited on pages 69 and 77)

[ELL09] Miguel Erazo, Yue Li, and Jason Liu. SVEET! A Scalable Virtualized Evaluation Environment for TCP. In *TridentCom’09: Proceedings of the 5th International Conference on Testbeds and Research Infrastructures for the Development of Networks & Communities*, Washington, DC, USA, April 6–8, 2009. ICST. (Cited on pages 14, 27, 42, 70, and 122)

[Fal99] Kevin Fall. Network Emulation in the Vint/NS Simulator. In *ISCC’99. Proceedings of the Fourth IEEE Symposium on Computers and Communications*, page 244, Sharm El Sheik, Red Sea, Egypt, July 6–8, 1999. IEEE Computer Society. (Cited on pages 16, 17, 31, 33, and 42)

[FG95] Francisco Javier Thayer Fábrega and Joshua D. Guttman. Copy on write. Technical report, The MITRE Corporation, <http://imps.mcmaster.ca/doc/copy-on-write.ps>, November 1, 1995. (Cited on page 114)

[FHL<sup>+</sup>05] Doug Freimuth, Elbert Hu, Jason LaVoie, Ronald Mraz, Erich Nahum,

Prashant Pradhan, and John Tracey. Server Network Scalability and TCP Offload. In *ATC'05: In Proceedings of the Annual Conference on USENIX Annual Technical Conference*, pages 15–15, Anaheim, CA, USA, April 10–15, 2005. USENIX Association. (Cited on page 72)

[FHN<sup>+</sup>04b] Keir Fraser, Steven Hand, Rolf Neugebauer, Ian Pratt, Andrew Warfield, and Mark Williamson. Safe Hardware Access with the Xen Virtual Machine Monitor. In *OASIS'04: Proceedings of the 1st Workshop on Operating System and Architectural Support for the on demand IT InfraStructure*, Boston, MA, USA, October 9–13, 2004. (Cited on page 35)

[Fis78] George Samuel Fishman. *Principles of Discrete Event Simulation*. John Wiley & Sons, Inc., New York, NY, USA, 1978. (Cited on page 32)

[Fuj89] Richard M. Fujimoto. Parallel Discrete Event Simulation. In *WSC'89: Proceedings of the 21st Winter Simulation Conference*, pages 19–28, Washington, DC, USA, December 4–6, 1989. ACM Press. (Cited on pages 13, 25, 26, 40, 42, and 70)

[Fur00] Stephen B. Furber. *ARM System-on-Chip Architecture (2nd Edition)*. Addison-Wesley Professional, August 2000. (Cited on page 35)

[GBC09] Pasquale Di Gennaro, Roberto Bifulco, and Roberto Canonico. Link Multiplexing in a Xen-based Network Emulation System. In *NGNM'09: Proceedings of the 6th International Workshop on Next Generation Networking Middleware (co-located with MANWEEK'09)*, pages 51–64, Venice, Italy, October 27–28, 2009. Multicon Lecture Notes no. 11, S. Figueira, M. Curado (Eds.), Multicon Verlag, Berlin. (Cited on page 65)

[Gc12] Greg Ganger and contributors. The DiskSim Simulation Environment. <http://www.pdl.cmu.edu/DiskSim/index.html>, 2012. (Cited on page 53)

[GFR<sup>+</sup>04] Fermín Galán, David Fernández, Javier Ruiz, Omar Walid, and Tomás de Miguel. Use of Virtualization Tools in Computer Network Laboratories. In *ITHET'04: Proceedings of the Fifth International Conference on Information Technology Based Higher Education and Training*, Istanbul, Turkey, May 31 – June 2, 2004. IEEE. (Cited on page 43)

[GHR09a] Andreas Grau, Klaus Herrmann, and Kurt Rothermel. Efficient and Scalable Network Emulation Using Adaptive Virtual Time. In *ICCCN'09: Proceedings of the 18th International Conference on Computer Communications and*

---

*Networks*, pages 1–6, San Fransisco, CA, USA, August 3–6, 2009. IEEE. (Cited on pages 14, 18, 19, 27, 43, 48, 51, 69, and 91)

[GHR09b] Andreas Grau, Klaus Herrmann, and Kurt Rothermel. Exploiting Emulation Testbeds for Security Experiments. In *Proceedings of the Workshop on Experimental platforms for Internet resilience, security and stability research (invited paper)*, pages 1–2, Brussels, Belgium, June 19, 2009. JRC European Commission. (Cited on page 31)

[GHR10] Andreas Grau, Klaus Herrmann, and Kurt Rothermel. NETplace: Efficient Runtime Minimization of Network Emulation Experiments. In *SPECTS'10: Proceedings of the International Symposium on Performance Evaluation of Computer and Telecommunication Systems*, pages 265–272, Ottawa, ON, Canada, July 11–14, 2010. IEEE Communications Society. (Cited on pages 18, 19, 20, 21, 48, 51, 52, 91, 93, 100, 103, 116, and 134)

[GHR11] Andreas Grau, Klaus Herrmann, and Kurt Rothermel. NETbalance: Reducing the Runtime of Network Emulation using Live Migration. In *ICCCN'11: Proceedings of the 20th International Conference on Computer Communications and Networks*, pages 1 – 6, Maui, HI, USA, July 31 – August 4, 2011. (Cited on pages 20, 21, 48, 94, 110, and 134)

[GHR12] Andreas Grau, Klaus Herrmann, and Kurt Rothermel. Scalable Network Emulation - The NET Approach. *Journal of Communications - Special Issue: Advances in Communications and Networking - II*, 7(1):3–16, January 2012. (Cited on pages 17, 18, 19, 20, 21, 43, 54, 93, and 94)

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. A Series of Books in the Mathematical Sciences (Victor Klee, ed.). W. H. Freeman and Company, 1979. (Cited on page 105)

[GLV<sup>+</sup>08] Diwaker Gupta, Sangmin Lee, Michael Vrable, Stefan Savage, Alex C. Snoeren, George Varghese, Geoffrey M. Voelker, and Amin Vahdat. Difference Engine: Harnessing Memory Redundancy in Virtual Machines. In *OSDI'08: Proceedings of the 8th USENIX Symposium on Operating System Design and Implementation*, San Diego, CA, USA, December 8–10, 2008. USENIX Association. (Cited on page 36)

[GMHR08] Andreas Grau, Steffen Maier, Klaus Herrmann, and Kurt Rothermel. Time Jails: A Hybrid Approach to Scalable Network Emulation. In *PADS'08:*

*Proceedings of the 22nd ACM/IEEE/SCS Workshop on Principles of Advanced and Distributed Simulation*, pages 1–8, Rome, Italy, June 3–6, 2008. IEEE Computer Society. (Cited on pages 13, 14, 16, 18, 19, 25, 26, 27, 32, 40, 43, 48, 51, 52, 54, 60, 69, 71, 72, and 91)

[GN98] Ivan Griffin and John Nelson. Linux Network Programming, Part 1. *Linux Journal*, 1998(46es):5, February 1998. (Cited on page 70)

[Gol73] Robert P. Goldberg. Architecture of Virtual Machines. In *Proceedings of the Workshop on Virtual Computer Systems*, pages 74–112, Cambridge, MA, USA, March 26–27, 1973. (Cited on page 35)

[Gol74] Robert P. Goldberg. Survey of Virtual Machine Research. *IEEE Computer*, 7(6):34–45, 1974. (Cited on page 35)

[Gra11] Andreas Grau. NET User Manual, December 2011. (Cited on pages 18, 48, and 95)

[GRL05] Shashi Guruprasad, Robert Ricci, and Jay Lepreau. Integrated Network Experimentation using Simulation and Emulation. In *TridentCom'05: Proceedings of the First International Conference on Testbeds and Research Infrastructures for the Development of Networks and Communities*, pages 204–212, Washington, DC, USA, February 22–25, 2005. IEEE Computer Society. (Cited on page 105)

[Gro06] Matthias Grossglauser. 10 Papers on Network Models. *Computer Communication Review*, 36(5):63–65, 2006. (Cited on pages 13 and 25)

[GSS<sup>+</sup>02] John Linwood Griffin, Jiri Schindler, Steven W. Schlosser, John C. Bucy, and Gregory R. Ganger. Timing-accurate Storage Emulation. In *FAST'02: Proceedings of the 1st USENIX Conference on File and Storage Technologies*, Monterey, CA, USA, January 28-30, 2002. USENIX Association. (Cited on page 53)

[GVV08] Diwaker Gupta, Kashi V. Vishwanath, and Amin Vahdat. Diecast: Testing distributed systems with an accurate scale model. In *NSDI'08: Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation*, pages 407–422, San Francisco, CA, USA, 2008. USENIX Association. (Cited on pages 17 and 43)

[GWS06] Andreas Grau, Harald Weinschrott, and Christopher Schwarzer. Evaluating the Scalability of Virtual Machines for Use in Computer Network Emulation. Case Study, Universität Stuttgart, Institute for Parallel and Distributed

---

Systems, Distributed Systems, Stuttgart, Germany, October 2006. (Cited on pages 35 and 58)

[GYM<sup>+</sup>06] Diwaker Gupta, Kenneth Yocum, Marvin McNett, Alex C. Snoeren, Amin Vahdat, and Geoffrey M. Voelker. To Infinity and Beyond: Time-Warped Network Emulation. In *NSDI'06: Proceedings of the 3rd ACM/USENIX Symposium on Networked Systems Design and Implementation*, pages 87–100, San Jose, CA, USA, May 8–10, 2006. USENIX Association. (Cited on pages 14, 17, 26, 27, 38, 39, 40, 42, 43, 54, 69, 71, 72, 121, and 129)

[Han09] Jacob Gorm Hansen. *Virtual Machine Mobility with Self-Migration*. PhD thesis, Department of Computer Science, University of Copenhagen, Copenhagen, Denmark, April 7, 2009. (Cited on page 110)

[HDG09] Michael R. Hines, Umesh Deshpande, and Kartik Gopalan. Post-Copy Live Migration of Virtual Machines. *ACM SIGOPS Operating Systems Review*, 43:14–26, July 2009. (Cited on page 111)

[Hem05] Stephen Hemminger. Network Emulation with NetEm. In *LCA'05: Proceedings of the Australia's 6th National Linux Conference (linux.conf.au)*, Canberra, Australia, April 18–23, 2005. (Cited on pages 16 and 32)

[Her05] Daniel Herrscher. *Emulation von Rechnernetzen zur Leistungsanalyse von verteilten Anwendungen und Netzprotokollen*. Dissertation, Universität Stuttgart : Sonderforschungsbereich SFB 627 (Nexus: Umgebungsmodelle für mobile kontextbezogene Systeme), Stuttgart, Germany, December 7, 2005. (Cited on pages 17, 19, 38, 45, 46, 54, 62, and 63)

[HGLP07] Wei Huang, Qi Gao, Jiuxing Liu, and D.K. Panda. High Performance Virtual Machine Migration with RDMA over Modern Interconnects. In *Cluster'07: Proceedings of the IEEE International Conference on Cluster Computing*, pages 11 –20, Austin, TX, USA, September 17–21, 2007. (Cited on pages 72 and 113)

[HH02] Jacob G. Hansen and Asger K. Henriksen. Nomadic Operating Systems. Master's thesis, Dept. of Computer Science, University of Copenhagen, Copenhagen, Denmark, December 10, 2002. (Cited on page 110)

[HK00] Bruce Hendrickson and Tamara G. Kolda. Graph partitioning models for parallel computing. *Parallel Computing*, 26(12):1519–1534, 2000. (Cited on page 103)

[HLW<sup>+</sup>11] Dongwoon Hahn, Ginnah Lee, Brenton Walker, Matt Beecher, and Padma

Mundur. Using Virtualization and Live Migration in a Scalable Mobile Wireless Testbed. *ACM SIGMETRICS Performance Evaluation Review*, 38:21–25, January 2011. (Cited on page 111)

[HR02] Daniel Herrscher and Kurt Rothermel. A Dynamic Network Scenario Emulation Tool. In *ICCCN'02: Proceedings of the 11th International Conference on Computer Communications and Networks*, pages 262–267, Miami, FL, USA, October 14–16, 2002. (Cited on pages 13, 16, 17, 19, 26, 31, 32, 43, 45, 48, 53, 54, 60, and 78)

[HRS<sup>+</sup>04] Mike Hibler, Robert Ricci, Leigh Stoller, Jonathon Duerig, Shashi Guuprasad, Tim Stack, Kirk Webb, and Jay Lepreau. Feedback-directed Virtualization Techniques for Scalable Network Experimentation. University of Utah Flux Group Technical Note FTN-2004-02, School of Computing, University of Utah, May 2004. (Cited on page 105)

[HRS<sup>+</sup>08] Mike Hibler, Robert Ricci, Leigh Stoller, Jonathon Duerig, Shashi Guuprasad, Tim Stack, Kirk Webb, and Jay Lepreau. Large-scale Virtualization in the Emulab Network Testbed. In *ATC'08: Proceedings of the USENIX 2008 Annual Technical Conference*, pages 113–128, Boston, MA, USA, June 22–27, 2008. USENIX Association. (Cited on pages 13, 25, 26, 37, 43, and 103)

[IEE06] IEEE. IEEE Standard for Local and Metropolitan Area Networks - Virtual Bridged Local Area Networks. *IEEE Std 802.1Q-2005 (Incorporates IEEE Std 802.1Q1998, IEEE Std 802.1u-2001, IEEE Std 802.1v-2001, and IEEE Std 802.1s-2002)*, pages 1 –285, May 19, 2006. (Cited on pages 19, 31, 46, 54, and 62)

[Int01] Intel Corporation. Intel 82093AA I/O Advanced Programmable Interrupt Controller (I/O APIC) at <http://www.intel.com/design/chipsets/specupdt/29071001.pdf>, January 2001. (Cited on page 60)

[Int10] Intel Corporation, Santa Clara, CA, USA. *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide, Part 1*, June 2010. (Cited on page 35)

[Int12] Intel Corporation. Intel 82580EB/82580DB Gigabit Ethernet Controller Datasheet at <http://www.intel.com/content/dam/doc/datasheet/82580-eb-db-gbe-controller-datasheet.pdf>, October 2012. (Cited on page 59)

[Jef85] David R. Jefferson. Virtual Time. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 7:404–425, July 1985. (Cited on page 40)

---

[JNVP06] Hyun-Wook Jin, Sundeep Narravula, Karthikeyan Vaidyanathan, and Dhabeleswar K. Panda. NemC: A Network Emulator for Cluster-of-Clusters. In *ICCCN'06: Proceedings of the 15th International Conference on Computer Communications and Networks*, pages 177–182, Arlington, VA, USA, October 9–11, 2006. (Cited on pages 43 and 60)

[JS82] David Jefferson and Henry A. Sowizral. Fast Concurrent Simulation Using the Time Warp Mechanism: Part I, Local Control. RAND Corporation <http://www.rand.org/pubs/notes/N1906>. Also available in print form., December 1982. (Cited on page 40)

[JX03] Xuxian Jiang and Dongyan Xu. vBET: a VM-Based Emulation Testbed. In *MoMeTools'03: Proceedings of the ACM SIGCOMM Workshop on Models, Methods and Tools for Reproducible Network Research*, pages 95–104, Karlsruhe, Germany, August 25, 2003. ACM Press. (Cited on pages 14, 26, and 43)

[KBKK06] Gunjan Khanna, Kirk Beaty, Gautam Kar, and Andryej Kochut. Application Performance Management in Virtualized Server Environments. In *NOMS'06: Proceedings of the 10th IEEE/IFIP Network Operations and Management Symposium*, pages 373 –381, Vancouver, BC, Canada, April 3–7, 2006. IEEE Communications Society. (Cited on page 110)

[KHS<sup>+</sup>03] Kenichi Kourai, Toshio Hirotsu, Koji Sato, Osamu Akashi, Kensuke Fukuda, Toshiharu Sugawara, and Shigeru Chiba. Secure and Manageable Virtual Private Networks for End-users. In *LCN'03: Proceedings of the 28th Annual IEEE International Conference on Local Computer Networks*, pages 385–394, Bonn/Königswinter, Germany, October 20–24, 2003. (Cited on pages 16, 36, 54, and 78)

[Kid05] Cameron Kiddle. *Scalable Network Emulation*. PhD thesis, University of Calgray, Calgary, AL, Canada, November 2005. AAINR03870. (Cited on pages 26 and 42)

[Kie09] Roman Lech Kierzkowski. Self-Organizing Distributed File System. Masterthesis, Politechnika Wrocławska, Wydział Informatyki i Zarządzania, Wrocław, Poland, January 2009. (Cited on pages 49 and 50)

[KK98a] George Karypis and Vipin Kumar. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, 1998. (Cited on pages 21, 104, 106, and 123)

[KK98b] George Karypis and Vipin Kumar. METIS, a Software Package for Parti-

tioning Unstructured Graphs and Computing Fill-Reduced Orderings of Sparse Matrices. Technical report, University of Minnesota, Department of Computer Science / Army HPC Research Center, 1998. (Cited on page 123)

[KKL<sup>+</sup>07] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguor. Kvm: the linux virtual machine monitor. In *OLS'07: Proceedings of the Ottawa Linux Symposium*, volume 1, pages 225–230, Ottawa, ON, Canada, June 27–30, 2007. (Cited on page 43)

[KLH<sup>+</sup>05] Yang-Suk Kee, Dionysios Logothetis, Richard Huang, Henri Casanova, and Andrew A. Chien. Efficient Resource Description and High Quality Selection for Virtual Grids. In *CCGrid'05: Proceedings of the Fifth IEEE International Symposium on Cluster Computing and the Grid*, pages 598–606, Cardiff, United Kingdom, May 9–12, 2005. IEEE Computer Society. (Cited on page 103)

[KP09] Stein Kristiansen and Thomas Plagemann. ns-2 Distributed Clients Emulation: Accuracy and Scalability. In *SIMUTools'09: Proceedings of the 2nd International Conference on Simulation Tools and Techniques*, pages 1:1–1:10, Rome, Italy, March 2–6, 2009. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering). (Cited on pages 17, 33, and 42)

[KS02] Michael Kozuch and M. Satyanarayanan. Internet Suspend/Resume. In *WMCSA'02: Proceedings of the Fourth IEEE Workshop on Mobile Computing Systems and Applications*, pages 40–, Callicoon, NY, USA, June 20–21, 2002. IEEE Computer Society. (Cited on pages 110 and 114)

[KSK09] Nikola Knežević, Simon Schubert, and Dejan Kostić. Towards a Cost-Effective Networking Testbed. *ACM SIGOPS Operating Systems Review*, 43(4):66–71, 2009. (Cited on page 100)

[KSU05] Cameron Kiddie, Rob Simmonds, and Brian Unger. Improving Scalability of Network Emulation through Parallelism and Abstraction. In *ANSS'05: Proceedings of the 38th Annual Simulation Symposium*, pages 119–129, San Diego, CA, USA, April 3–7, 2005. SCS/ACM. (Cited on pages 42 and 122)

[KW00] Poul-Henning Kamp and Robert N. M. Watson. Jails: Confining the omnipotent root. In *Sane'00: Proceedings of the 2nd International SANE Conference*, Maastricht, The Netherlands, May 22–25, 2000. (Cited on pages 16, 34, 37, 54, and 78)

[LC03] Xin Liu and Andrew A. Chien. Traffic-based Load Balance for Scalable

---

Network Emulation. In *SC'03: Proceedings of the ACM/IEEE Conference on Supercomputing*, page 40, Phoenix, AZ, USA, November 15–21, 2003. ACM. (Cited on page 105)

[LC04] Xin Liu and Andrew A. Chien. Realistic Large-Scale Online Network Simulation. In *SC'04: Proceedings of the ACM/IEEE Conference on Supercomputing*, Pittsburgh, PA, USA, November 4–6, 2004. IEEE Computer Society. (Cited on pages 14, 27, and 103)

[LFG<sup>+</sup>01] Benyuan Liu, Daniel R. Figueiredo, Yang Guo, Jim Kurose, and Don Towsley. A Study of Networks Simulation Efficiency: Fluid Simulation vs. Packet-level Simulation. In *INFOCOM'01: Proceedings of the Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies*, volume 3, pages 1244–1253 vol.3, Anchorage, AK, USA, April 22–26, 2001. IEEE. (Cited on pages 26 and 42)

[Liu08a] Jason Liu. A Primer for Real-Time Simulation of Large-Scale Networks. In *ANSS'08: Proceedings of the 41st Annual Simulation Symposium*, volume 0, pages 85–94, Ottawa, ON, Canada, April 13–16, 2008. IEEE Computer Society. (Cited on pages 13 and 25)

[Liu08b] Jason Liu. Immersive Real-Time Large-Scale Network Simulation: A Research Summary. In *IPDPS'08: Proceedings of the IEEE International Symposium on Parallel and Distributed Processing*, Miami, FL, USA, April 14–18, 2008. IEEE. (Cited on pages 13, 25, 26, and 70)

[LLH09] Jason Liu, Yue Li, and Ying He. A Large-scale Real-time Network Simulation Study using PRIME. In *WSC'09: Proceedings of the Winter Simulation Conference*, pages 797–806, Austin, TX, USA, December 13–16, 2009. (Cited on pages 16, 17, 31, 33, and 42)

[LLXC05] Yi Liu, Yanping Li, Kaiping Xiao, and Huali Cui. Mapping Resources for Network Emulation with Heuristic and Genetic Algorithms. In *PDCAT'05: Proceedings of the Sixth International Conference on Parallel and Distributed Computing Applications and Technologies*, pages 670–674, Dalian, China, December 5–8, 2005. IEEE Computer Society. (Cited on pages 14, 27, 103, and 105)

[LR09] Jason Liu and Raju Rangaswam. Model-Driven Network Emulation with Virtual Time Machine. Technical Report TR-2009-03-0, Florida International University, 2009. (Cited on page 42)

[LUSG04] Joshua LeVasseur, Volkmar Uhlig, Jan Stoess, and Stefan Götz. Unmodified Device Driver Reuse and Improved System Dependability via Virtual Machines. In *OSDI'04: Proceedings of the 6th Conference on Symposium on Operating Systems Design and Implementation*, pages 2–2, San Francisco, CA, USA, December 6–8, 2004. USENIX Association. (Cited on page 58)

[LZW<sup>+</sup>08] Yingwei Luo, Binbin Zhang, Xiaolin Wang, Zhenlin Wang, Yifeng Sun, and Haogang Chen. Live and Incremental Whole-System Migration of Virtual Machines Using Block-Bitmap. In *Cluster'08: Proceedings of the IEEE International Conference on Cluster Computing*, pages 99–106, Tsukuba, Japan, September 29 – October 1, 2008. (Cited on page 111)

[Mai11] Steffen Maier. *Scalable Computer Network Emulation Using Node Virtualization and Resource Monitoring*. Dissertation, Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Stuttgart, Germany, February 16, 2011. (Cited on pages 14, 17, 19, 36, 37, 45, 62, 63, and 69)

[MAS10] Rick McGeer, David G. Andersen, and Stephen Schwab. The Network Testbed Mapping Problem. In *TridentCom'10: Proceedings of the 6th International Conference on Testbeds and Research Infrastructures for the Development of Networks and Communities*, Berlin, Germany, May 18–20, 2010. ICST. (Cited on page 103)

[MBB00] Tony McGregor, Hans-Werner Braun, and Jeff Brown. The NLAMR network analysis infrastructure. *Communications Magazine, IEEE*, 38(5):122–128, May 2000. (Cited on page 122)

[MGWR07] Steffen Maier, Andreas Grau, Harald Weinschrott, and Kurt Rothermel. Scalable Network Emulation: A Comparison of Virtual Routing and Virtual Machines. In *ISCC'07: Proceedings of the IEEE Symposium on Computers and Communications*, pages 395–402, Aveiro, Portugal, July 1–4, 2007. IEEE Computer Society. (Cited on pages 16 and 36)

[MHR07] Steffen Maier, Daniel Herrscher, and Kurt Rothermel. Experiences with Node Virtualization for Scalable Network Emulation. *Computer Communications*, 30(5):943–956, March 8, 2007. (Cited on pages 26, 31, 43, and 45)

[Mis86] Jayadev Misra. Distributed Discrete-Event Simulation. *ACM Computing Surveys*, 18(1):39–65, 1986. (Cited on page 42)

[MKK08] Andrey Mirkin, Alexey Kuznetsov, and Kir Kolyshkin. Containers check-

---

pointing and live migration. In *OLS'08: Proceedings of the Linux Symposium*, Ottawa, ON, Canada, July 23–26, 2008. (Cited on pages 110 and 127)

- [MLMB01] Alberto Medina, Anukool Lakhina, Ibrahim Matta, and John Byers. BRITE: An Approach to Universal Topology Generation. In *MASCOTS'01: Proceedings of the International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunications Systems*, Cincinnati, OH, USA, August 15–18, 2001. IEEE Computer Society. (Cited on pages 50, 98, and 122)
- [MRBV05] Priya Mahadevan, Adolfo Rodriguez, David Becker, and Amin Vahdat. MobiNet: A Scalable Emulation Infrastructure for Ad Hoc and Wireless Networks. In *WiTMeMo'05: Proceedings of the International Workshop on Wireless Traffic Measurements and Modeling (In conjunction with MobiSys)*, Seattle, WA, USA, June 5, 2005. (Cited on page 64)
- [MvdPJ09] Abraham Mukosi Mukwevho, John Andrew van der Poll, and Robert Mark Jolliffe. A Virtual Integrated Network Emulator on XEN (viNEX). In *SIMU-Tools'09: Proceedings of the 2nd International Conference on Simulation Tools and Techniques*, pages 3:1–3:7, Rome, Italy, March 2–6, 2009. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering). (Cited on page 43)
- [NCTP07] Ranjit Noronha1, Lei Chai1, Thomas Talpey, and Dhabaleswar K. Panda1. Designing NFS with RDMA for Security, Performance and Scalability. In *ICPP'07: Proceedings of the International Conference on Parallel Processing*, page 49, Xi-An, China, September 10–14, 2007. IEEE Computer Society. (Cited on page 114)
- [Net12a] Netgear GSM7352S-200 Datasheet. <http://netgear.com/service-provider/products/switches/fully-managed-switches/gsm7352s-200.aspx>, 2012. (Cited on page 64)
- [Ope12] OpenVZ. <http://openvz.org>, 2012. (Cited on pages 54, 78, 83, and 127)
- [OSSN02] Steven Osman, Dinesh Subhraveti, Gong Su, and Jason Nieh. The Design and Implementation of Zap: A System for Migrating Computing Environments. *ACM SIGOPS Operating Systems Review*, 36:361–376, December 2002. (Cited on pages 110 and 112)
- [PACR03] Larry Peterson, Tom Anderson, David Culler, and Timothy Roscoe. A Blueprint for Introducing Disruptive Technology into the Internet. *ACM*

*SIGCOMM Computer Communication Review*, 33(1):59–64, 2003. (Cited on pages 41 and 42)

[Pak08] Frederik Pakai. Adaptation of the Time Dilation Factor in a Time Virtualized Emulation Environment. Diploma thesis, no. 2795, Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Stuttgart, Germany, December 31, 2008. (Cited on page 69)

[PBR<sup>+</sup>08] Simon Peter, Andrew Baumann, Timothy Roscoe, Paul Barham, and Rebecca Isaacs. 30 Seconds is Not Enough! A Study of Operating System Timer Usage. *ACM SIGOPS Operating Systems Review*, 42(4):205–218, 2008. (Cited on page 61)

[PF07] Herbert Pötzl and Marc E. Fiuczynski. Linux-VServer - Resource Efficient OS-Level Virtualization. In *OLS'07: Proceedings of the Ottawa Linux Symposium*, Ottawa, ON, Canada, June 27–30, 2007. (Cited on pages 34 and 36)

[PH08] David A. Patterson and John L. Hennessy. *Computer Organization and Design: the Hardware/Software Interface (4th Edition)*. Morgan Kaufmann, November 2008. (Cited on page 35)

[PR08] Maurizio Pizzonia and Massimo Rimondini. Netkit: Easy Emulation of Complex Networks on Inexpensive Hardware. In *TridentCom'08: Proceedings of the 4th International Conference on Testbeds and Research Infrastructures for the Development of Networks & Communities*, pages 1–10, Innsbruck, Austria, March 18–20, 2008. ICST. (Cited on page 43)

[Pre92] Roger S. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw Hill, 1992. (Cited on pages 13 and 25)

[RAL03] Robert Ricci, Chris Alfeld, and Jay Lepreau. A Solver for the Network Testbed Mapping Problem. *ACM SIGCOMM Computer Communication Review*, 33(2):65–81, 2003. (Cited on pages 14, 27, 103, and 105)

[RDR11] Stamatia Rizou, Frank Dürr, and Kurt Rothermel. Fulfilling End-to-End Latency Constraints in Large-scale Streaming Environments. In *IPCCC'11: Proceedings of the 30th IEEE International Performance Computing and Communications Conference*, pages 1–8. IEEE Xplore, November 2011. (Cited on page 49)

[RFA99] G. F. Riley, R. M. Fujimoto, and M. H. Ammar. A Generic Framework for Parallelization of Network Simulations. In *MASCOTS'99: Proceedings of the 7th International Symposium on Modeling, Analysis and Simulation of*

---

*Computer and Telecommunication Systems*, pages 128–135, College Park, MD, USA, October 24–28, 1999. IEEE Computer Society. (Cited on pages 42 and 70)

[RGSX06] Himanshu Raj, Ivan Borissov Ganev, Karsten Schwan, and Jimi Xenidis. Self-Virtualized I/O: High Performance, Scalable I/O Virtualization in Multi-core Systems. Technical report, Georgia Institute of Technology, 2006. (Cited on page 36)

[RH11] Fabian Romero and Thomas J. Hacker. Live Migration of Parallel Applications with OpenVZ. In *AINA'11: Proceeding of the 25th IEEE International Conference on Advanced Information Networking and Applications*, pages 526 –531, Biopolis, Singapore, March 22–25, 2011. (Cited on page 110)

[RHV06] Joseph F. Ruscio, Michael A. Heffner, and Srinidhi Varadarajan. DejaVu: Transparent User-Level Checkpointing, Migration, and Recovery for Distributed Systems. In *SC'06: Proceedings of the ACM/IEEE Conference on Supercomputing*, page 158, Tampa, FL, USA, November 11–17, 2006. ACM Press. (Cited on page 110)

[Ril03] George F. Riley. The Georgia Tech Network Simulator. In *MoMeTools'03: Proceedings of the ACM SIGCOMM Workshop on Models, Methods and Tools for Reproducible Network Research*, pages 5–12, Karlsruhe, Germany, August 25, 2003. ACM Press. (Cited on pages 13, 25, and 26)

[Riz97] Luigi Rizzo. Dummynet: A Simple Approach to the Evaluation of Network Protocols. *ACM SIGCOMM Computer Communication Review*, 27(1):31–41, 1997. (Cited on pages 13, 16, 26, 31, and 32)

[ROLV06] Robert Ricci, David Oppenheimer, Jay Lepreau, and Amin Vahdat. Lessons from Resource Allocators for Large-scale Multiuser Testbeds. *ACM SIGOPS Operating Systems Review*, 40(1):25–32, 2006. (Cited on page 103)

[Ros04] Robert Rose. Survey of System Virtualization Techniques. Technical report, Oregon State University, 2004. (Cited on page 35)

[Rus08] Rusty Russell. virtio: Towards a De-Facto Standard For Virtual I/O Devices. *ACM SIGOPS Operating Systems Review*, 42:95–103, July 2008. (Cited on page 58)

[SBP07] Peter Jay Salzman, Michael Burian, and Ori Pomerant. *The Linux Kernel Module Programming Guide*. Open Software License, May 2007. (Cited on page 83)

## Bibliography

---

[Sch00] Brian Keith Schmidt. *Supporting Ubiquitous Computing with Stateless Consoles and Computation Caches*. PhD thesis, Stanford University, Stanford, CA, USA, August 2000. (Cited on pages 16, 37, and 78)

[Sch10] Markus Schirmer. Distributed Emulation of Mobile Applications in TVEE. Study thesis, no. 2273, Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Stuttgart, Germany, November 3, 2010. (Cited on page 138)

[Sch11a] Markus Schirmer. Scalable Emulation of Network Links between Virtual Nodes. Diploma thesis, no. 3132, Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Stuttgart, Germany, June 30, 2011. (Cited on pages 51 and 62)

[Sch11b] Frank Schuh. Entwicklung eines skalierbaren Emulationsrahmenwerks für mobile Knoten. Diploma thesis, no. 3170, Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Stuttgart, Germany, October 11, 2011. (Cited on page 95)

[SCP<sup>+</sup>02] Constantine P. Sapuntzakis, Ramesh Chandra, Ben Pfaff, Jim Chow, Monica S. Lam, and Mendel Rosenblum. Optimizing the Migration of Virtual Computers. *ACM SIGOPS Operating Systems Review*, 36:377–390, December 2002. (Cited on page 111)

[SD04] Ananth I. Sundararaj and Peter A. Dinda. Towards Virtual Networks for Virtual Machine Grid Computing. In *VM'04: Proceedings of the 3rd conference on Virtual Machine Research And Technology Symposium*, pages 14–14, San Jose, CA, USA, May 6–7, 2004. USENIX Association. (Cited on page 41)

[SGRC10] Jason Sonnek, James Greensky, Robert Reutiman, and Abhishek Chandra. Starling: Minimizing Communication Overhead in Virtualized Computing Platforms Using Decentralized Affinity-Aware Migration. In *ICPP'10: Proceedings of the 39th International Conference on Parallel Processing*, pages 228 –237, San Diego, CA, USA, September 13–16, 2010. IEEE Computer Society. (Cited on page 110)

[SHR10] Stephan Schuhmann, Klaus Herrmann, and Kurt Rothermel. Efficient Resource-Aware Hybrid Configuration of Distributed Pervasive Applications. In *Proceedings of the 8th International Conference on Pervasive Computing (Pervasive 2010); Helsinki, Finland, May 17-20, 2010*, volume 6030

---

of *Lecture Notes in Computer Science*, pages 373–390. Springer-Verlag, May 2010. (Cited on page 49)

[SKM08] Aameek Singh, Madhukar Korupolu, and Dushmanta Mohapatra. Server-Storage Virtualization: Integration and Load Balancing in Data Centers. In *SC'08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, pages 53:1–53:12, Austin, TX, USA, November 15–21, 2008. IEEE/ACM. (Cited on page 110)

[SKS02] Naranjan G. Shivaratri, Phillip Krueger, and Mukesh Singhal. Load Distributing for Locally Distributed Systems. *Computer*, 25(12):33–44, 2002. (Cited on page 111)

[SM79] L. H. Seawright and R. A. MacKinnon. VM/370 – A study of multiplicity and usefulness. *IBM Systems Journal*, 18(1):4–17, 1979. (Cited on pages 16 and 34)

[SMWA04] Neil Spring, Ratul Mahajan, David Wetherall, and Thomas Anderson. Measuring ISP Topologies with Rocketfuel. *IEEE/ACM Transactions on Networking*, 12:2–16, February 2004. (Cited on page 122)

[SPF<sup>+</sup>07] Stephen Soltesz, Herbert Poetzl, Marc E. Fiuczynski, Andy Bavier, and Larry Peterson. Container-based Operating System Virtualization: A Scalable, High-performance Alternative to Hypervisors. In *EuroSys'07: Proceedings of the 2nd ACM European Conference on Computer Systems*, pages 275–287, Lisbon, Portugal, March 21–23, 2007. ACM Press. (Cited on pages 16 and 36)

[SPYH03] Osamu Sato, Richard Potter, Mitsuharu Yamamoto, and Masami Hagiya. UML Scrapbook and Realization of Snapshot Programming Environment. In *ISSS'03: Software Security: Theories and Systems (Second Mext-NSF-JSPS International Symposium)*, volume 3233 of *Lecture Notes in Computer Science*, pages 281–295. Springer, Tokyo, Japan, November 4–6, 2003. (Cited on pages 43 and 112)

[Ste08] Illya Stepanov. *Using geographic models in the simulation of mobile communication - Verwendung von Geographischen Modellen in der Simulation von mobiler Kommunikation*. Dissertation, Universität Stuttgart : Sonderforschungsbereich SFB 627 (Nexus: Umgebungsmodelle für mobile kontextbezogene Systeme), Stuttgart, Germany, October 2008. (Cited on pages 95, 98, and 122)

[SU03] Rob Simmonds and Brian W. Unger. Towards Scalable Network Emulation.

*Computer Communications*, 26(3):264–277, 2003. (Cited on pages 16, 17, 31, 33, and 42)

[SVL01] Jeremy Sugerman, Ganesh Venkitachalam, and Beng-Hong Lim. Virtualizing I/O Devices on VMware Workstation’s Hosted Virtual Machine Monitor. In *ATC’01: Proceedings of the USENIX Annual Technical Conference*, pages 1–14, Boston, MA, USA, June 25–30, 2001. USENIX Association. (Cited on page 57)

[TH98] Joe Touch and Steve Hotz. The X-Bone. In *Globecom’98: Proceedings of the Global Internet Mini-Conference*, Sydney, Australia, November 8–12, 1998. (Cited on page 41)

[TLC85] Marvin M. Theimer, Keith A. Lantz, and David R. Cheriton. Preemptable Remote Execution Facilities for the V-System. In *SOSP’85: Proceedings of the tenth ACM Symposium on Operating Systems Principles*, pages 2–12, Orcas Island, WA, USA, December 1–4, 1985. ACM Press. (Cited on page 111)

[TRR08] Michael Tüxen, Irene Rüngeler, and Erwin P. Rathgeb. Interface connecting the INET simulation framework with the real world. In *SIMUTools’08: Proceedings of the 1st International Conference on Simulation Tools and Techniques for Communications, Networks and Systems*, pages 40:1–40:6, Marseille, France, March 3–7, 2008. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering). (Cited on pages 16, 17, 31, 33, and 42)

[VGR<sup>+</sup>11] Alexandra Voit, Andre Grund, Daniel Reichelt, Davide Casciato, Dimitrij Pankratz, Florian Pfleiderer, Hannes Todenhagen, Hendrik Glück, Johannes Wettinger, Julian Trischler, Markus Funk, and Ruben Mayer. NET Captain Help, July 2011. (Cited on pages 18 and 98)

[VMC<sup>+</sup>05] Michael Vrable, Justin Ma, Jay Chen, David Moore, Erik Vandekieft, Alex C. Snoeren, Geoffrey M. Voelker, and Stefan Savage. Scalability, Fidelity, and Containment in the Potemkin Virtual Honeyfarm. In *SOSP’05: Proceedings of the twentieth ACM Symposium on Operating Systems Principles*, pages 148–162, Brighton, UK, October 23–26, 2005. ACM Press. (Cited on page 36)

[VYW<sup>+</sup>02] Amin Vahdat, Ken Yocom, Kevin Walsh, Priya Mahadevan, Dejan Kostić, Jeffrey Chase, and David Becker. Scalability and Accuracy in a Large-Scale Network Emulator. In *OSDI’02: Proceedings of the 5th ACM/USENIX Symposium on Operating Systems Design and Implementation*, pages 1–12, San Francisco, CA, USA, October 21–24, 2002. ACM Press. (Cited on page 111)

---

*sium on Operating System Design and Implementation*, Boston, MA, USA, December 9–11, 2002. USENIX Association. (Cited on pages 14, 26, 43, and 105)

[Wal02] Carl A. Waldspurger. Memory Resource Management in VMware ESX Server. *ACM SIGOPS Operating Systems Review, Special Issue: Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI'02)*, 36:181–194, December 2002. (Cited on pages 35 and 36)

[WCSG04] Andrew Whitaker, Richard S. Cox, Marianne Shaw, and Steven D. Gribble. Constructing Services with Interposable Virtual Hardware. In *NSDI'04: Proceedings of the 1st Symposium on Networked Systems Design and Implementation*, pages 13–13, San Francisco, CA, USA, March 29–31, 2004. USENIX Association. (Cited on page 36)

[WK02] Shie-Yuan Wang and Hsiang-Tsung Kung. A New Methodology for Easily Constructing Extensible and High-Fidelity TCP/IP Network Simulators. *Computer Networks*, 40(2):205–315, 2002. (Cited on pages 17, 39, 42, and 70)

[WLR02] Marc H. Willebeek-LeMair and Anthony P. Reeves. Strategies for Dynamic Load Balancing on Highly Parallel Computers. *IEEE Transactions on Parallel and Distributed Systems*, 4(9):979–993, 2002. (Cited on page 111)

[WLS<sup>+</sup>02] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An Integrated Experimental Environment for Distributed Systems and Networks. In *OSDI'02: Proceedings of the 5th ACM/USENIX Symposium on Operating System Design and Implementation*, pages 255–270, Boston, MA, USA, December 9–11, 2002. USENIX Association. (Cited on page 105)

[WSG02] Andrew Whitaker, Marianne Shaw, and Steven D. Gribble. Scale and Performance in the Denali Isolation Kernel. *ACM SIGOPS Operating Systems Review*, 36(SI):195–209, 2002. (Cited on pages 14, 26, and 36)

[WSHW08] Elias Weingärtner, Florian Schmidt, Tobias Heer, and Klaus Wehrle. Synchronized Network Emulation: Matching prototypes with complex simulations. In *HotMetrics'08: Proceedings of the First Workshop on Hot Topics in Measurement & Modeling of Computer Systems*, Annapolis, MD, USA, June 6, 2008. ACM, SIGMETRICS PER. (Cited on pages 16, 31, 40, 42, and 70)

[WSvL<sup>+</sup>11] Elias Weingärtner, Florian Schmidt, Hendrik vom Lehn, Tobias Heer, and Klaus Wehrle. SliceTime: A platform for scalable and accurate network emulation. In *NSDI'11: Proceedings of the 8th USENIX Symposium on Net-*

*worked Systems Design and Implementation*, Boston, MA, USA, March 30 – April 1, 2011. USENIX Association. (Cited on page 42)

[WSVY07] Timothy Wood, Prashant Shenoy, Arun Venkataramani, and Mazin Yousif. Black-box and Gray-box Strategies for Virtual Machine Migration. In *NSDI'07: Proceedings of the 4th USENIX Conference on Networked Systems Design & Implementation*, pages 17–17, Cambridge, MA, USA, April 11–13, 2007. USENIX Association. (Cited on page 110)

[WTLS<sup>+</sup>09] Timothy Wood, Gabriel Tarasuk-Levin, Prashant Shenoy, Peter Desnoyers, Emmanuel Cecchet, and Mark D. Corner. Memory Buddies: Exploiting Page Sharing for Smart Colocation in Virtualized Data Centers. In *VEE'09: Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 31–40, Washington, DC, USA, March 11–13, 2009. ACM. (Cited on page 110)

[WVLW09] Elias Weingärtner, Hendrik Vom Lehn, and Klaus Wehrle. A performance comparison of recent network simulators. In *ICC'09: Proceedings of the IEEE International Conference on Communications*, pages 1287–1291, Dresden, Germany, June 14–18, 2009. IEEE Press. (Cited on page 33)

[WWG08] Jian Wang, Kwame-Lante Wright, and Kartik Gopalan. XenLoop: A Transparent High Performance Inter-VM Network Loopback. In *HPDC'08: Proceedings of the 17th ACM International Symposium on High Performance Distributed Computing*, Boston, MA, USA, June 23–27, 2008. ACM. (Cited on pages 36 and 59)

[XOP11] Xavier Besseron, Xiangyong Ouyang, Raghunath Rajachandrasekar and Dhabaleswar K. Panda. High Performance Pipelined Process Migration with RDMA. In *CCGrid'11: Proceedings of the 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 314–323, Newport Beach, CA, USA, May 23–26, 2011. (Cited on pages 72 and 113)

[YBYW08] Ben-Ami Yassour, Muli Ben-Yehuda, and Orit Wasserman. Direct Device Assignment for Untrusted Fully-Virtualized Virtual Machines. Technical report, IBM Research, 2008. (Cited on pages 57 and 58)

[YED<sup>+</sup>03] Ken Yocum, Ethan Eade, Julius Degesys, David Becker, Jeff Chase, and Amin Vahdat. Toward Scaling Network Emulation using Topology Partitioning. In *MASCOTS'03: In Proceedings of the 11th IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer Telecommunications and Networks*, pages 1–10, Boston, MA, USA, October 12–14, 2003. IEEE Computer Society.

---

*munications Systems*, pages 242 – 245, Orlando, FL, USA, October 12–15, 2003. IEEE Computer Society. (Cited on page 103)

[YFS03] Lingyun Yang, Ian Foster, and Jennifer M. Schopf. Homeostatic and Tendency-based CPU Load Predictions. In *IPDPS'03: Proceedings of the International Parallel and Distributed Processing Symposium*, page 9 pp., Nice, France, April 22–26, 2003. IEEE Computer Society. (Cited on page 115)

[YP11] Srikanth B. Yeginath and Kalyan S. Perumalla. Efficiently Scheduling Multi-Core Guest Virtual Machines on Multi-Core Hosts in Network Simulation. In *PADS'11: Proceedings of the 25th ACM/IEEE/SCS Workshop on Principles of Advanced and Distributed Simulation*, pages 1 –9, Nice, France, June 14–17, 2011. IEEE Computer Society. (Cited on page 70)

[YYK08] Tetsuya Yoshida, Hiroshi Yamada, and Kenji Kono. FoxyLargo: Slowing Down CPU Speed with a Virtual Machine Monitor for Embedded Time-Sensitive Software Testing. In *IWVT'08: Proceedings of the International Workshop on Virtualization Technology*, Beijing, China, June 21–25, 2008. (Cited on page 53)

[Zay87] Edward R. Zayas. Attacking the Process Migration Bottleneck. In *SOSP'87: Proceedings of the eleventh ACM Symposium on Operating Systems Principles*, pages 13–24, Austin, TX, USA, November 8–11, 1987. ACM Press. (Cited on page 111)

[Zho11] Kai Zhou. Development of a Load Model for Distributed Systems. Diploma thesis, no. 3082, Universität Stuttgart, Fakultät Informatik, Elektrotechnik und Informationstechnik, Stuttgart, Germany, March 23, 2011. (Cited on page 99)

[ZN03] Pei Zheng and Lionel M. Ni. EMPOWER: A Network Emulator for Wireless and Wireline Networks. In *INFOCOM'03: Proceedings of the 14th Annual Joint Conference of the IEEE Computer and Communications Societies*, San Francisco, CA, USA, March 30 – April 3, 2003. IEEE. (Cited on pages 14, 27, 64, and 103)

[ZN11] Yuhao Zheng and David M. Nicol. A Virtual Time System for OpenVZ-Based Network Emulations. In *PADS'11: In Proceedings of the IEEE Workshop on Principles of Advanced and Distributed Simulation*, pages 1–10, Nice, France, June 14–17, 2011. IEEE Computer Society. (Cited on pages 17 and 39)