Scalable Network Emulation - The NET Approach

Andreas Grau, Klaus Herrmann, Kurt Rothermel
Universität Stuttgart, Institute of Parallel and Distributed Systems (IPVS), Stuttgart, Germany
Email: {grau,herrmann,rothermel}@ipvs.uni-stuttgart.de

Abstract— **Network emulation is an efficient method for evaluating distributed applications and communication protocols by combining the benefits of real world experiments and network simulation. The process of network emulation involves the execution of connected virtual nodes running the software under test in a controlled environment. Our *Network Emulation Testbed* (NET) achieves high scalability by combining efficient node virtualization and adaptive virtual time.**

In this paper, we provide an overview of our system. First, we introduce our efficient emulation architecture. Second, we present our approaches (*NETplace* and *NETbalance*) to minimize the runtime of the network experiments. The idea of *NETplace* is to minimize the load of the testbed by calculating an initial placement of virtual nodes onto the testbed nodes. During the runtime of the experiment *NETbalance* adapts this placement to changed resource requirements of the software under test. Finally, we introduce *NETcaptain*, a graphical user interface to setup, control and visualize network experiments.

Index Terms— **Scalable Network Emulation, Placement, Migration, Node Virtualization, Virtual Time**

I. Introduction

Performance evaluation is an integral part of the software development process. In the field of distributed systems, there are mainly three types of performance evaluation methodologies: network simulation [1]–[3], real-world testbeds [4], and network emulation [5]–[7]. Network emulation, 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. These networks are modeled by connecting routers and hosts running instances of the *software under test* (SuT). Using our emulation tool *NETshaper* [6], the parameters of these network links are adjustable and include bandwidth, delay, and loss rate. In our Network Emulation Testbed (NET), the experiments are executed on a cluster of commodity PC-nodes. To enable large-scale experiments, we run multiple instances of the SuT (encapsulated in so-called *virtual nodes*) on each of these PC-nodes (called *physical nodes*).

The CPU-load of a physical node directly depends on the number of virtual nodes running on it. Overload of a physical node may bias the results of an experiment, because, for example, messages between virtual nodes experience additional, undesired delays. Virtual time [8], which decouples the time experienced by the virtual nodes from the real time, allows for avoiding such overload situations. Slowing down the virtual time reduces the execution speed of an experiment and, thus, reduces the load on the physical nodes. However, a constant slowdown increases the runtime of an experiment with fluctuating resource requirements unnecessarily. Therefore, NET provides an extended concept, called *Adaptive Virtual Time* [9]. By adapting the speed of the virtual clocks to the current system load, NET prevents system overload as well as system underload of the physical nodes in the testbed. When the system load is low, the virtual time is accelerated to speed up the experiment and when the system load is high, it is slowed down to prevent overload.

Using adaptive virtual time, the runtime of experiments is determined by the physical node with the highest load. Minimizing this load minimizes the runtime of the experiment. In order to reach this goal, we have developed NETplace [10]. The basic idea is to calculate, based on an average experiment load including the network and CPU usage of virtual nodes, an initial placement of virtual nodes onto physical nodes that minimizes the load of the physical nodes. In order to compare the experiment runtime of different placements of virtual nodes during this calculation, we have developed an accurate testbed model [10] to predict the load of the physical nodes based on a placement of virtual nodes.

Due to the principle of an initial placement, experiments with scenarios with varying and unknown load result in suboptimal runtime. To achieve also in these scenarios minimal runtime, we developed an extended approach called *NETbalance* [11]. *NETbalance* monitors the load of the virtual nodes to detect load changes and trigger the migration of virtual nodes during the

This paper is based on "NETbalance: Reducing the Runtime of Network Emulation using Live Migration" by A. Grau, K. Herrmann, and K. Rothermel, which appeared in the Proc. of the 20th Int. Conf. on Computer Communications and Networks (ICCCN), USA, 2011. © 2011 IEEE; "NETplace: Efficient Runtime Minimization of Network Emulation Experiments" by A. Grau, K. Herrmann, and K. Rothermel, which appeared in the Proc. of the Int. Symp. on Performance Evaluation of Computer and Telecommunication Systems (SPECTS), Canada, 2010; © 2010 IEEE; "Efficient and Scalable Network Emulation using Adaptive Virtual Time," by A. Grau, K. Herrmann, and K. Rothermel, which appeared in the Proc. of the 18th Int. Conf. on Computer Communications and Networks (ICCCN), USA, 2009; © 2009 IEEE.

This work was supported in part by the Deutsche Forschungsgemeinschaft (German Research Foundation) grant DFG-GZ RO 1086/9-3.

Manuscript received August 15, 2011; revised October 15, 2011; accepted November 25, 2011.

doi:10.4304/jcm.7.1.3-16

experiment runtime. 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. A new placement is only deployed if the resulting speed-up outweighs the cost for the migrations.

Finally, the time required for running networks experiments is influenced by the time to design and setup the experiment. To minimize also this time, we have developed *NETcaptain*. *NETcaptain* is a plug-in for the common used integrated development platform *Eclipse* and, provides an graphical user interface for modeling the network topology, deploying the software under test, gathering the generated log files, monitoring and visualizing the state of the virtual nodes and links. The virtual topology can be manually modified during the running experiment or using a powerful scripting engine.

The contributions of this paper are as follows:

1) an overview of NET system and the interaction of the NET components,
2) extended evaluation results of the NET components
3) introduction of our control component *NETcaptain*

The remainder of this paper is structured as follows. In Section II we present related work of the NET system. The architecture of NET, including the basic concepts of node and time virtualization are introduced in Section III. In Section IV, we present our Testbed model followed by our concepts used for the inital placement and the dynamic reconfiguration of the testbed. The section closes with a brief discussion of the management software for our system. Detailed evaluation results of the NET components are discussed in Section V. In Section VI, we give a summary and conclusions.

II. Related Work

1) Node Virtualization: The resources of a testbed node can be partitioned on different layers. The spectrum ranges from emulating the entire hardware [12] including processor architecture over virtual machines (VMs) [13] and virtual protocol stacks [14] to process memory separation [15] where each process runs in exclusive virtual memory as is provided by common operating systems.

VM-based emulators [7], [8], [16] allow for running a complete operating system as software under test. However, this flexibility introduces a significant computation and memory overhead. Emulators [17], [18] like NET [6] based on a lightweight virtualization like virtual protocol stacks, avoid this overhead by only virtualizing network specific resources of the operating system.

2) Time Virtualization: A constant slowdown [8], [19] of the virtual clocks can be used to avoid overloading the testbed. However, varying resource requirements of the software under test leads to temporary unused resources and, thus, an extension of the experiment runtime.

Hybrid systems [1] combine network simulation [2], [20] and time-virtualized network emulation [18] to run unmodified implementations. However, a constant clock rate is used here too. The necessary synchronization in the simulation produces additional overhead.

Weingärtner et. al. [16] used a barrier synchronization to conservatively synchronize multiple virtual machines to a simulation framework. This allows for running the experiment with adaptive virtual time. However, the VM-based node virtualization increases the overhead introduced by the synchronization schema.

Emulation of arbitrarily powerful virtual resources can be achieved by adapting the Linux protocol stack to use virtual time instead of real time. While Wang et. al. [21] use only a simulation framework running on a single physical node, dONE [17] uses a distributed simulation environment. In contrast to our system, dONE only supports testing of application layer implementations using the BSD socket interface.

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

3) Initial Testbed Configuration: The assignment of virtual nodes to physical nodes is essential to achieve good emulation performance. In the following, we give a brief outline of related assignment strategies.

In real-time testbeds [4], [5], [7] the assignment is constraint by limited processing capabilities of the physical nodes and the bandwidth and delay of the links in between. This *constraint satisfaction problem* can be solved using evolutionary algorithms [22], [23], backtracking [24], or bin-backing [25]. Since the experiments run in real-time, the goal of the approaches is to find a solution that satisfies the constraints. Therefore, these approaches can hardly be used to minimize the experiment runtime of a virtual time-based network emulator.

In parallel computing [26] as well as in network simulation [27] the execution time is minimized by minimizing the total bandwidth between physical nodes using graph partitioning frameworks [28]. However, the basic assumption of these approaches is not valid in our system. For minimal runtime we need to minimize the bandwidth per physical node and not the overall bandwidth.

4) Dynamic Testbed Reconfiguration: Up to now, the migration of virtual nodes is not used by any network emulator to reduce the experiments runtime. Approaches from other areas using similar concepts are investigated for their applicability to our problem in the following.

The migration of virtual nodes is similar to task migration to achieve load balancing [29], [30]. In contrast to our problem, here, the placement of one task does not influence the execution costs of another task.

The migration time can be minimized by transferring the memory state before suspending [31] or after resuming [32] the process. However, applied to network emulation, in both approaches the state is transfered in parallel to the running experiment and, therefore, will increase the CPU usage which leads to a slower experiment execution.

Therefore, none of the existing approaches can minimize the runtime of experiments.

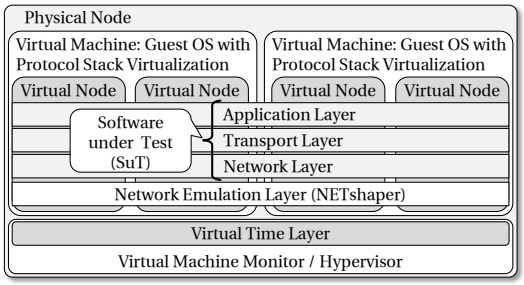

Figure 1. TVEE architecture: Multiple virtual protocol stack instances inside one virtual machine

III. System Architecture

A. Efficient Node Virtualization

In order to support highly scalable network experiments in NET, we have developed the *Time Virtualized Emulation Environment* (TVEE) [6]. As shown in Figure 1, TVEE is based on two building blocks: *node virtualization* and *time virtualization*. Node virtualization [6] allows for executing multiple virtual nodes running the *Software under Test* (SuT) on a single physical node. Time virtualization [9] provides a real-time-independent *virtual time* to the virtual nodes. The quotient of real time and virtual time is called *time dilation factor* τ. Slowing down the clocks of the virtual nodes by τ reduces the load of the physical nodes by the same factor. A closed-loop controller [9] running on a central coordinator adapts τ to the load of the physical nodes. This adaption maximizes the execution speed of an experiment without overloading the physical nodes. A detailed discussion of the adaption is provided in the following section.

In order to provide virtual time transparently to the SuT, we make use of the virtual machine (VM) abstraction [13]. For maximum efficiency, we run one VM on each CPU of a physical node [10]. *Virtual protocol stacks* [14], [33] allow for creating virtual nodes [6] by partitioning the operating system running inside the virtual machine. All virtual nodes running in the same VM share a common operating system. This approach minimizes the memory overhead per virtual node and allows virtual nodes to communicate efficiently using reference passing. Using our network emulation tool *NETshaper* [34], we are able to build arbitrary network topologies with user-definable parameters (bandwidth, delay, loss rate). Since the network emulation is located on the *Data Link* layer, our emulation architecture supports evaluations on the *Network*, *Transport*, and *Application* layer.

B. Adaptive Virtual Time

To adapt the virtual clock rate to the resource demand of the experiment we introduce the *Time Dilation Factor* τ. Equation 1 shows how the virtual time (R_v) and the real time (R_r) are related by means of the τ.

$$R_v = 2^{-\frac{\tau}{10}} \cdot R_r \tag{1}$$

In order to allow an efficient implementation using integer arithmetic, we use (in contrast to Gupta et. al. [8]) a logarithmic relation between the rate of virtual time and the real time. Using only integer arithmetics, we can adjust the rate of the virtual clock with a step width of about 7%. This granularity is sufficient for the adaptation algorithm that we will introduce in Section III-B.2. In addition, without a logarithmic relation, this granularity depends on the virtual clock rate. For fast rates the granularity is coarse and it increases with slower rates.

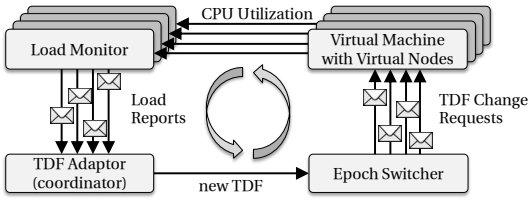

Figure 2. TDF adaptation schema

In order to perform the adaption of τ, we propose the concept of epoch-based virtual time [6]. The experiment is divided in epochs of different length where τ is constant within each epoch. Whenever the resource demand changes, an epoch switch is triggered to adapt τ.

Figure 2 shows the TDF adaptation schema. Each physical node of the emulation system runs a *load monitor* that monitors the node's load and reports it to a *central coordinator* who calculates the overall system load. The overall load is defined as the maximum over all individual node load values. Since each node may run multiple VMs (one VM per CPU), the load of a node is defined as the load of the maximum loaded CPU of the node. This definition is chosen to ensure that no CPU of the physical node is overloaded at any time.

Using the overall load, the coordinator determines a new τ and initiates an epoch switch. The *epoch switcher* is used to distribute the new τ 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.

1) Distributed Load Monitoring: The *load monitor* is used to measure the load of a physical node and report the load to the *TDF adaptor*. The virtual machine monitor (VMM) provides a per virtual machine statistic, which counts the number of used CPU cycles $c(t)$. Requesting this value at 2 points in time ($c(t_1)$ and $c(t_2)$) allows for calculation of the load $l = \frac{c(t_2)-c(t_1)}{t_2-t_1}$. To meter the time between the measurements with a sub-microsecond granularity, we use the time stamp counter register of 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 scalability. To limit the amount of load reports, we use 3 mechanisms: *adaptive sampling*, *threshold-based discretization* and *hysteresis-based state changes*. These mechanisms effectively reduce communication overhead for reporting

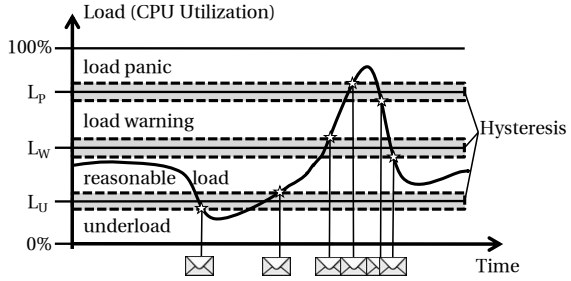

Figure 3. Load Monitoring Thresholds

substantially.

Adaptive sampling adjusts the length of the sampling interval (time period between two consecutive load reports) to the currently used τ. For a higher τ (slower virtual time) a longer sampling interval is chosen. The ratio 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. The effect is that as a system increases in size (number of virtual nodes) and the load increases as a result, the message overhead resulting from load reports decreases. Therefore, we increase the sampling interval linearly with τ.

Threshold-based discretization maps the possible load values of a physical node to the 4 states *load panic*, *load warning*, *reasonable load*, and *underload* (see Figure 3) using 3 thresholds (L_P, L_W, and L_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* indicates that there are unused resources and, therefore, virtual time could be accelerated. Analogous, 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 L_P and L_W are used to differentiate between slight and heavy *load*. The different reaction on these states is described in the next section.

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 (see Figure 3).

2) TDF Adaptation: 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 τ 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 τ again to speed up virtual time.

After each adjustment, the algorithm needs to wait for

input: $state$, τ_{prev}

```
1  while true do
2      if state != reasonable_load then
3          if state = load_panic then
4              setTDF(τ_prev + S_l)
5          else if state = load_warning then
6              setTDF(τ_prev + S_s)
7          else if state = underload then
8              setTDF(τ_prev - S_s)
9          end
10         sleep T_s
11     else if state = reasonable_load then
12         setTDF(τ_prev - S_s)
13         sleep T_l
14     end
15 end
```

Algorithm 1. TDF Adaption Process

feedback from the load monitor to see whether the load is back in the state *reasonable load*. Due to the adaptive sampling of the load monitor, the time until the feedback arrives depends on the current value of τ. Therefore, we dynamically adjust the waiting time (T_s) to the half of the used sampling interval.

In case of temporarily constant resource demands, the utilization can keep steady at any level between the L_U and L_W thresholds in the state *reasonable load*. For good resource usage, however, the system utilization should be near the L_W threshold. Therefore, we decrease τ 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 oscillation around L_W, these adjustments introduce an insignificant overhead.

Our evaluation shows that the introduced algorithm has a good reaction to changes of resource requirements despite the fact that it is rather simple.

3) Epoch Switching: After determining τ 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 change of τ must be as small as possible. Since the time to compute the new value of τ is negligibly small, we need to minimize the time for transmitting load reports and TDF change requests. In addition, realistic 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 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

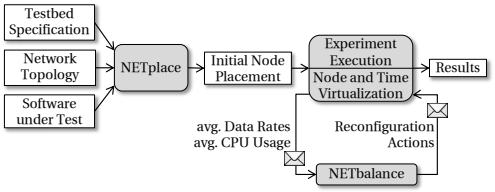

Figure 4. Experiment execution workflow

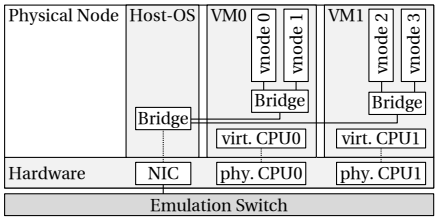

Figure 5. Architecture of NET using multiple CPUs

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 200μs and has a 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 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 theses 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 2ms and message loss can be prevented with a very high probability.

IV. Testbed Configuration

Running an experiment using NET follows the workflow depicted in Figure 4. Based on the testbed specification (number of physical nodes, CPUs per physical node, CPU capacity), the network topology and the expected average resource requirements of the SuT, NETplace [10] calculates an initial placement that minimizes the experiment runtime. 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.

In this basic workflow, resource requirements of the SuT deviating from the average may lead to temporary suboptimal placements which leads to an extended experiment runtime. NETbalance [11] extends this workflow by an additional compontent to monitor the resource requirements of virtual nodes, and to adapt, if required, the placement of virtual nodes to changing requirements. The base of the initial placement and the dynamic reconfiguration is a testbed model to determine the load of physical nodes based on the placement of virtual nodes.

Finally, in order to execute a network experiment efficiently, we have developed an graphical user interface *NETcaptain* to specify the network experiment and to control and visualize the experiment execution.

In the following, we first introduce the testbed model. Then we discuss our approaches for the initial placement and the dynamic placement of virtual nodes. Finally, we introduce the graphical user interface of NET.

A. Testbed Model

A network experiment in NET consists of a set N of virtual nodes. Each virtual node $i \in N$ runs a ***SuT*** which consumes λ_i CPU cycles and transmits β_{ij} data to virtual node j per time unit. The experiment runs for θ_{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 λ_i and β_{ij} is provided by the experimenter or can be gathered during the execution of a scenario. 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 P of physical nodes. Each physical node $p \in P$ is equipped with a set C_p of CPUs and runs $|C_p|$ virtual machines. Each CPU can perform ν_{CPU} CPU cycles per real time unit. We identify each VM by addressing the physical node p and the CPU $c \in C_p$ that is assigned to the VM (p, c). The set of VMs is named V. As shown in Figure 5, the host operating system (*host-os*) of each physical node enables communication between the virtual machines. We define the placement of the virtual nodes onto the virtual machines as a function $\phi : i \mapsto (p, c)$. In order to calculate a placement that minimizes experiment runtime θ_{real} (real time), we need a cost model μ for the testbed. As shown in Equation 2, based on the placement of the virtual nodes, this cost model allows for the calculation of the physical nodes' load . This load consists of the load $\Lambda_p^{\text{host-os}}$ of the host-os and the load $\Lambda_{p,c}^{\text{vm}}$ of the VMs running on the physical node. The unit of Λ is CPU cycles per time.

$$\mu : (\phi, p) \mapsto (\Lambda_p^{\text{host-os}}, \Lambda_{p,1}^{\text{vm}}, \ldots, \Lambda_{p,|C_p|}^{\text{vm}}) \qquad (2)$$

In order to define μ, we first discuss how virtual nodes can communicate. We distinguish 3 types of links between virtual nodes: *intra-vm*, *inter-vm* and *inter-pnode links*.

- *Intra-vm links* are the most efficient way to connect two virtual nodes. Here, both virtual nodes are running inside the same VM. 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 (see Table 1).
- *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

	VM_{tx}	VM_{rx}	Host-OS	PNode	Testbed
intra-vm	+	+	−	+	+
inter-vm	++	++	++	++	++
inter-pnode	++	++	+++	++/++	+++

Table 1. Cost matrix κ (the number of + denotes the amount of load generated by a link to the components; no load is indicated by −).

some additional overhead. Evaluation shows that inter-vm links are about 10 times more expensive than intra-vm links.

- *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. Evaluation shows, that these type of links introduce the highest overhead and cause about 2 times more load than the inter-vm links and 20 times more load than the intra-vm links.

PNode (see Table 1) defines the total load generated on a physical node: $VM_{tx} + VM_{rx} + 2 * \text{Host-OS}$. For inter-pnode links, this load is distributed to 2 physical nodes. *Testbed* defines the generated load throughout the testbed.

Based on the link type, the data rate β_{ij} of a link between two virtual nodes i and j, and the cost matrix κ (see Table 1), Equation 3 defines the generated load of the host-os $\Lambda_p^{\text{host-os}}$.

$$\Lambda_p^{\text{host-os}} = \sum_{\substack{i,j \in N \\ \phi(i)=(p,c) \\ \phi(j)=(p',c')}} \beta_{ij} * \begin{cases} \kappa_{\text{intra-vm}}^{\text{host-os}} & \text{if } p = p' \wedge c = c', \\ \kappa_{\text{inter-vm}}^{\text{host-os}} & \text{if } p = p' \wedge c \neq c' \\ \kappa_{\text{inter-pnode}}^{\text{host-os}} & \text{if } p \neq p' \end{cases} \tag{3}$$

The load of a VM $\Lambda_{p,c}^{\text{vm}}$ is calculated using Equation 4, where $\Lambda_{p,c}^{\text{VM}_{tx}}$ and $\Lambda_{p,c}^{\text{VM}_{rx}}$ are defined analogous to $\Lambda_p^{\text{host-os}}$. The VM running the virtual node i that transmits data is loaded by $\kappa_{\text{VM}_{tx}}$ and the VM running the virtual node j that receives data is loaded by $\kappa_{\text{VM}_{rx}}$. The SuT additionally loads the VMs by λ_i and λ_j, respectively.

$$\Lambda_{p,c}^{\text{vm}} = \Lambda_{p,c}^{\text{VM}_{tx}} + \Lambda_{p,c}^{\text{VM}_{rx}} + \sum_{i \in N \wedge \phi(i)=(p,c)} \lambda_i \tag{4}$$

We can calculate the load of the physical CPUs $\Lambda_{p,c}$ using Equation 5. Since the virtual CPUs of the VMs 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. 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 $\frac{1}{|C_p|}$ of the overall load generated on the physical node p.

$$\Lambda_{p,c} = \max\left(\Lambda_{p,c}^{vm}, \frac{1}{|C_p|} * (\Lambda_p^{\text{host-os}} + \sum_{c' \in C_p} \Lambda_{p,c'}^{\text{vm}})\right) \tag{5}$$

Knowing the load of each CPU, we can define the experiment runtime using Equation 6. The maximum loaded CPU has to execute $\Lambda_{p,c} * \theta_{\text{virtual}}$ CPU cycles during the experiment. The number of cycles divided by the speed of the CPUs ν_{CPU} results in the experiment runtime θ_{real}.

$$\theta_{\text{real}} = \max_{p \in P, c \in C_p} (\Lambda_{p,c}) * \frac{\theta_{\text{virtual}}}{\nu_{\text{CPU}}} \tag{6}$$

The cost matrix κ 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 κ 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 κ in Table 1 by minimizing the difference between the measured load and the load calculated based on κ and the measured data rates.

B. Initial Testbed Configuration

Next, we present our approaches to calculate the placement ϕ that minimizes the experiment runtime. We first propose 2 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.

1) Edge-Cut-based Approaches: In order to place virtual nodes using edge-cut-based approaches, the virtual network is modeled as a weighted graph γ. The weight of a vertex v_i, which represents a virtual node, is defined as the load of the virtual nodes' SuT λ_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 [28] 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 on the same virtual machine. In the following, we extend this approach to minimize the runtime of network emulation experiments.

a) 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 λ_i plus the sum of all virtual links of the virtual node times the intra-vm costs (see Equation 7).

$$v_i = \lambda_i + \sum_{j \in N} (\beta_{ij} + \beta_{ji}) * \kappa_{\text{intra-vm,PNode}} \tag{7}$$

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.

b) 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 γ two times. During the first run, we partition the graph γ into

$|P|$ partitions $\{\gamma_1, \ldots, \gamma_{|P|}\}$. Each partition is assigned to a single physical node. For each physical node p, we again partition the corresponding subgraph γ_p into $|C_p|$ partitions $\{\gamma_{p,1}, \ldots, \gamma_{p,|C_p|}\}$. The partitions $\gamma_{p,c}$, generated in this second run, are assigned to the virtual machines running on the physical node p.

2) Greedy Approaches: A simple greedy approach 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 (initial cluster member, *ICM*) to each cluster. Second, we assign the remaining nodes randomly, one by one to the minimum loaded cluster (*MLC*). Due to the random selection, the load is distributed uniformly to the physical nodes.

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 heuristic 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 $\max_{c \in C_p}(\Lambda_{p,c})$ is minimal. From the VMs running on this physical node, we select the VM (p, c) where the load $\Lambda^{\text{vm}}_{p,c}$ is minimal. Assigning additional virtual nodes to this VM will unlikely increase the experiment time. We call the cluster of this VM the minimal loaded cluster (*MLC*).

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 an subsequent optimization phase. This optimization is used for the greedy and the edge-cut-based approach.

3) Optimization of Node Assignments: 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 minimizes the experiment runtime. The proposed optimization is performed after the edge-cut based approaches as well as the greedy approaches.

Let $\{t_1, \ldots, t_{|V|}\} = a$ be an assignment of virtual nodes N to VMs V, where $t_i \cap t_j = \emptyset$ and $\bigcup_{i=1}^{|V|} t_i = N$. We also call such an assignment as a *state*. In order to optimize an assignment, we use hill climbing to minimize a cost function $\zeta(a)$. We define ζ later in this section. During each round, we generate *neighboring states* $\{a'_1, \ldots, a'_e\}$ by removing a virtual node n form a cluster i and assigning n to a cluster j: $\{t_1, \ldots, t_i \setminus \{n\}, \ldots, t_j \cup \{n\}, \ldots, t_{|V|}\}$. 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 $\zeta(a'_{\text{b}}) < \zeta(a)$ the optimization continues with the state a'_{b}.

Due to the large number of neighboring states $O(|N| * |V|)$, rating these states requires a large effort. 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 only one random neighboring state $a'_{\text{r}} \in \{a'_1, \ldots, a'_e\}$. In case $\zeta(a'_{\text{r}}) < \zeta(a)$ the optimization continues with a'_{r}, otherwise it continues with a. The optimization terminates if we cannot find any neighboring state with lower costs. Generally, we could limit the number of state changes to provide hard time limits to the placement algorithm. However, experiments have shown that the local optima was always reached after a short period of time.

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 (see Equation 8) 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.

$$\zeta = \begin{pmatrix} \zeta_1 \\ \zeta_2 \end{pmatrix} = \begin{pmatrix} \max_{p \in P \wedge c \in C_p}(\Lambda_{p,c}) \\ \sum_{p \in P \wedge c \in C_p}(\Lambda_{p,c})^2 \end{pmatrix} \tag{8}$$

Equation 9 is used to compare a state a and a neighboring state a'. Here, we first compare ζ_1 because reducing the load of the maximum loaded CPU results directly in a reduced experiment runtime. In case ζ_1 is equal for both states (third line of Equation 9), we compare ζ_2.

$$\zeta(a') < \zeta(a) \Leftrightarrow \begin{cases} \text{true} & \text{if } \zeta_1(a') < \zeta_1(a), \\ \text{false} & \text{if } \zeta_1(a') > \zeta_1(a), \\ \zeta_2(a') < \zeta_2(a) & \text{if } \zeta_1(a') = \zeta_1(a) \end{cases} \tag{9}$$

C. Dynamic Testbed Reconfiguration

In order to minimize the experiment runtime in scenarios with varying resource requirements, NETbalance [11] adapts the placement of virtual nodes to changing resource requirements of virtual nodes. These changes trigger the re-calculation of the virtual nodes' placement. We adopt the concept of *live migration* [35] to transform the current placement into the optimized placement by migrating virtual nodes between virtual machines.

After changing the placement, the experiment runs with an increased execution speed. However, this speed-up 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 speed-up. Our assumption is that we can predict the 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, the migration cost model, 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 30s in advance [36]. As reported by Yang et. al. [37] 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 schema. 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 τ, 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 20KB[1] per physical node. Significant changes in the load of virtual nodes trigger the calculation of a new placement ϕ'. To calculate ϕ', the coordinator adapts the current placement ϕ to the changed load. Using the *testbed cost model* [10] developed for NETplace, we can calculate the time dilation factor for the current placement τ_ϕ 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*. This cost model (discussed at the end of this section) includes the costs of transferring the virtual node's state between the VMs and of modifying the virtual topology.

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 $T_o \ll T_p$. After T_o, we abort the simulated annealing-based algorithm used for minimizing the cost function χ:

$$\chi = [(T_p - T_o) \cdot \tau_{\phi'} + T_r] - (T_p - T_o) \cdot \tau_\phi \qquad (10)$$

χ represents the reduction of the experiment runtime in the prediction window T_p. The runtime of the current placement ϕ is subtracted from the runtime of the new placement ϕ', 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 ϕ' and for executing the transformation $\phi \rightarrow \phi'$, ϕ' takes effect over the time window $T_p - T_o$. If χ is negative, then the transition to ϕ' will result in a speed-up of the experiment and NETbalance configures the system accordingly. If, however, χ is positive, then ϕ' performs worse than ϕ and we keep the configuration ϕ. Thus, the experiment runtime cannot increase through the optimization.

The value of T_o determines the performance of NETbalance. Increasing T_o allows more time for finding better placements. At the same time, however, the time $T_p - T_o$ left for actually running the better configuration ϕ' gets smaller. In our evaluation, we investigate in 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 to exceed the reconfiguration costs; otherwise, it is discarded. This effectively serves as a hysteresis, avoiding constant re-configuration with minimal gain.

After calculating a new placement of virtual nodes, we need to enforce the changes to the placement by migrating virtual nodes. Each of these virtual nodes is migrated from a virtual machine VM_{src} to a virtual machine VM_{dst}. We stop the experiment before we transfer the state of a virtual node. The incurred reconfiguration cost T_r is estimated using a migration cost model that we introduce at the end of this section.

To ensure that the re-placement does not influence the emulation results, the migration of virtual nodes must be transparent to the SuT. Therefore, we stop the experiment synchronously on the physical nodes which includes two phases. First, by setting the time dilation factor to infinity, the virtual clocks are stopped. 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 stopped, we change the placement of virtual nodes. For this, we adopt the concepts of the ZAP system [38]. First, we create a snapshot of a virtual node using check-pointing. 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. The state of the virtual node is then transferred to VM_{dst} and restored thereafter. The network interfaces of the restored virtual nodes are reattached to the virtual topology.

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_{dst}. Due to the typical size of a file system, copying all files introduces a large overhead. To avoid this overhead, we store the file system of a virtual node on a central server[2]. 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 virtual nodes. All virtual nodes

[1] Scenario with 4 network links per virtual node

[2] The central file server could be implemented by a cluster of file servers in case of performance bottlenecks

use a common *Copy-on-Write* file system to share these files. To minimize the overhead, we are using hard links to make shared files available on 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. The synchronization can run in parallel to the migration and does not contribute to T_r since the files are written back while the virtual nodes are suspended, and we are assuming only small changes to the file system, implying a fast synchronization and a negligible effect on the node's state size. In the case of larger changes, the state of the virtual node grows. In our evaluation, we show the effect 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. Since we are using suspend/resume migration [39], the reconfiguration time T_r is defined as follows:

$$T_r = T_{\text{suspend}} + T_{\text{migrate}} + T_{\text{change-topology}} + T_{\text{resume}} \quad (11)$$

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

The migrations of virtual nodes running in the same VM are performed sequentially. However, since the migration of a virtual node generates only load on VM_{src} and VM_{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.

D. Integrated Experiment Platform

Figure 6 shows our graphical user interface *NETcaptain*. The integration into the development platform Eclipse allows for directly evaluating of distributed application using our network emulator. *NETcaptain* supports to 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) are integrated. The mobility and connectivity of wireless-connected mobile nodes is based on trace-files and several radio propagation models (e.g. ray-tracing). The experiments can be live-visualized using a flexible visualization engine and controlled by powerful scripting engine.

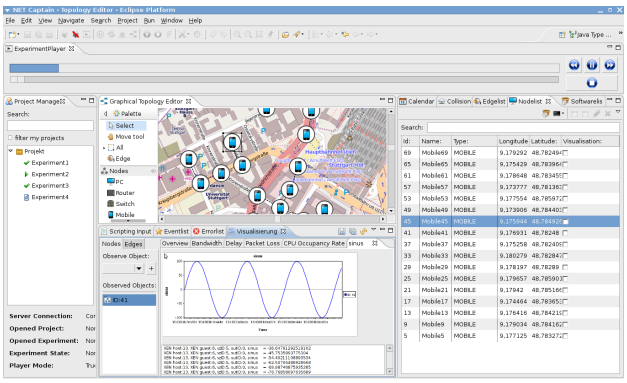

Figure 6. NETcaptain: GUI of the NET system

V. Evaluation

The evaluation of NET contains 5 steps: We show the efficiency and emulation accuracy of our virtualization architecture (1). We discuss the effectiveness of the adaptive time virtualization (2). The accuracy of our testbed model is presented (3). We show the performance of the initial node placement (4), and finally, we show the accuracy and effectiveness of the dynamic reconfiguration (5).

A. Network Emulation

1) Bandwidth Emulation: First, we measure if the emulation layer is able to enforce a configured bandwidth faithfully. Therefore, we set up a scenario with 2 connected virtual nodes in two variations. One variant uses a single physical node hosting both virtual nodes and the other uses two physical nodes with one virtual node each. We configure the link bandwidth with different values ranging from 64 kbps to 100 Gbps and no additional delay. To measure the maximum throughput of the link, we use the netperf tool [40] in UDP mode. It generates load according to configured send and receive buffers of 64 kB and an Ethernet MTU of 1500 Bytes.

As shown in Figure 7, the measured throughput corresponds to the configured bandwidth. Note that, due to the used hardware, for high speed links an emulation running at real-time is not possible. Therefore, we increased the τ to avoid overloading of emulation nodes.

2) Delay Emulation: Next, we examine if the emulation tool faithfully reproduces configured delays. Again, the scenario consists of 2 connected virtual nodes. The link has a bandwidth of 100 Mbps and a variable delay between 1 ms and 100 ms. We use the ping tool to measure the round trip time between the virtual nodes. Variations of this scenario use one or two physical nodes to host virtual nodes on same or different physical nodes respectively. We also vary the τ. As shown in Figure 8, the delays are emulated accurately.

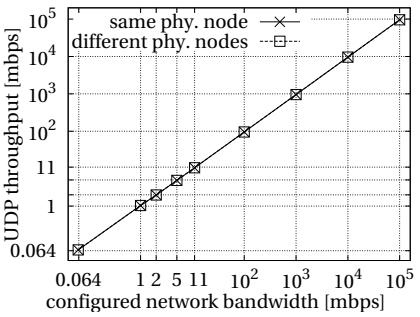

Figure 7. Accuracy of bandwidth emulation

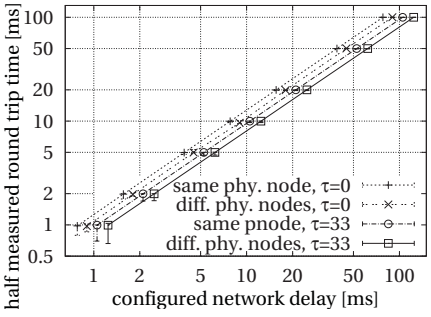

Figure 8. Accuracy of delay emulation

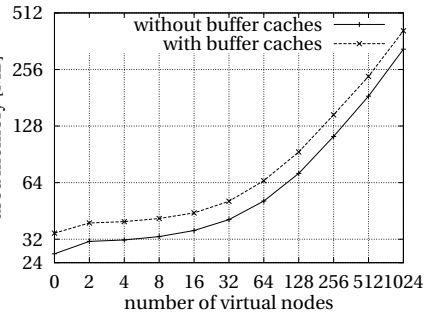

Figure 9. Memory overhead

3) Memory Consumption: As outlined during the discussion of the TVEE architecture, the scalability of an emulation solution 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 the same network. Figure 9 shows the required memory usage.

The memory usage consists of two components. One constant amount for the base system including Linux kernel requiring about 27 MB and a constant per virtual node memory usage of about 300 kB. In comparison, a virtual node-based on XEN requires at minimum 6 MB of memory [13]. As shown in Figure 9, 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, dom0 (64 MB), and virtual machine.

B. Adaptive Virtual Time

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 ensure unbiased results. The determined thresholds of the load monitor are: L_U=50, L_W=70, and L_P=90. The adaptive sampling interval ranges from 5ms for τ=0 to 200ms for τ=100. Adjustments of τ with a step width S_s of 1 and S_l of 20 give best results for the adaptation of τ.

To quantify the achieved level of resource consumption, we are emulating a chain of routers routing 2 TCP flows. The test system consists of two physical nodes. On the first one, 2 virtual nodes are running the TCP sender and receiver of the first flow F_f. This flow is routed through the chain of routers with different lengths. The routers run on the second physical node. Additionally, one link of the router chain is used by a second flow F_b. The emulated network between the virtual nodes has a bandwidth of 1Gbps except for the first and last link which have 100Mbit. During each experiment, we run the TCP flow F_b for 20s of virtual time. After 5s we run the flow F_f for 10s and measure the achieved throughput. In addition, the resource usage on both physical nodes is measured. For each router chain length, the experiment is repeated 50 times.

Figure 10 shows the CPU utilization of the physical node running a chain of 32 routers. The time axis

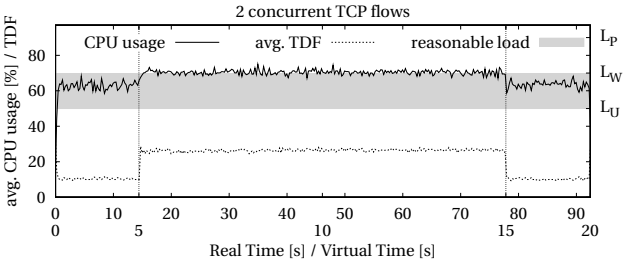

Figure 10. Load-based TDF adaptation

has two sales: the upper scale is the real time and the lower scale the virtual time. Running only the flow F_b requires the system to run with $\tau \approx 10$ to keep the CPU utilization inside the *reasonable load* range. Running flow F_f between 5s and 15s of virtual time increases the resource requirements. In order to prevent overload, the system automatically adapts $\tau \approx 27$. As flow F_f stops, the system adapts τ back to the original value.

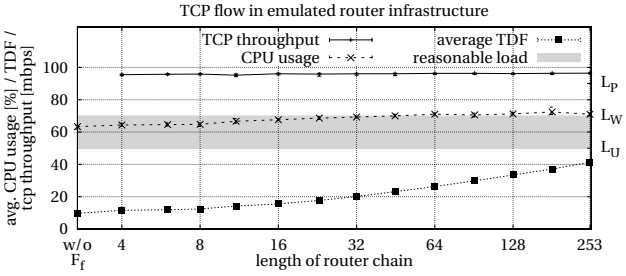

Figure 11. Effectiveness of TDF adaptation

Figure 11 shows the measured results for different numbers of routers, which are: the achieved TCP throughput, the load of the physical node running the router chain, and the average value of τ. Although these measurements have different scales, we show them in a single graph to increase comparability. For comparison, we have also included the results for the experiment without flow F_f. The TCP throughput allows to rate the quality of the emulation by comparing the 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 96Mbps throughput and, therefore, we can conclude that the emulation results are not biased.

As shown in the figure, for up to 8 routers the resource utilization mainly results from flow F_b. As the number of routers is increased, the resource requirements for flow F_f increase likewise. Since each router basically does

	VM_{tx}	VM_{rx}	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	7.78/7.31	15.09

Table 2. Costs matrix κ for a emulation testbed consisting of quad-core machines (unit of κ is $\frac{cycles}{byte}$)

the same, the load increases linearly with the number of routers. At a length of about 11 routers, the flow F_f consumes a significant amount of CPU and, therefore, the system needs to slow down the virtual time.

The gray area in Figure 11 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 values of τ results in a small sampling interval (see III-B.1) 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 τ. However, the sensitivity is required to prevent overload situations.

C. Testbed Model

In order to evaluate the accuracy of the cost model, we determine the cost matrix κ using the method introduced in Section IV-A. A router chain loaded by a single TCP connection 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.

Table 2 shows the costs matrix κ for the 3 types of links. Column VM_{tx} shows the costs of a VM running a virtual node that transmits data. Column VM_{rx} shows the costs of a VM running a virtual node that receives data. The costs of the host-os is shown in Column *Host-OS*. Column *PNode* indicates the summed up costs per physical node. The last column (*Testbed*) shows sum of the costs generated by a link throughout the testbed. The values show that inter-vm links generate almost 10 times more load than intra-vm links. Inter-pnode links generate 2 times more load than inter-vm and about 20 times more load than intra-vm links. However, in case of inter-pnode links, the load is distributed to 2 physical nodes and, therefore, the physical node costs (*PNode*) are slightly lower than the costs generated by inter-vm links.

In order to evaluated the accuracy of our testbed model, we emulated a second scenario shown in Figure 12. The scenario consists of 2500 video sensor nodes arranged in a regular grid. The one node on

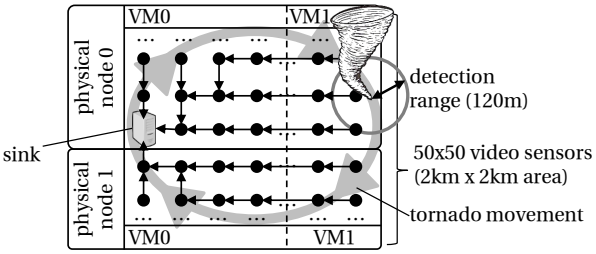

Figure 12. Video Sensor Network

the left acts as a sink. The routes in the scenario are established using geometric routing. The nodes are running a software that monitors the a moving tornado. In case the tornado is in sight, the sensor sends a 10Mbps stream to the sink, otherwise a 32kbps stream. The nodes are distributed to 2 physical nodes each with the 2 virtual machines. The measured data rate at the sink and the value of τ are visualized in Figure 13a.

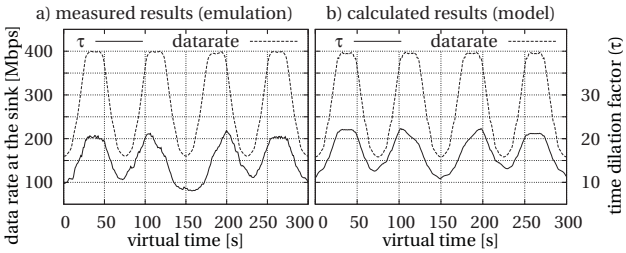

Figure 13. Video Sensor Network

Figure 13b shows for the same setup the calculated date rate and the required time dilation factor τ using the testbed model. Comparing the calculated and measured results, we can conclude that our model allows for accurate load calculation.

D. Initial Testbed Configuration

In order to evaluate the experiment runtimes of the placement approaches, we use in the following 9 scenarios. (1) A *Wlan* model with 1,892 nodes randomly distributed over a city with wireless communication. Due to the road network, the node density varies, resulting in nodes with a high and low number of links. (2,3,4) Scenarios (*NetworkMap* [41], *Internet* [42] and *AT&T* [43]) based on a snapshots of Internet topology 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 [18], [44]. We use 20 campuses with a total of 5,480 nodes. (8,9) The final scenarios are generated by the topology generator BRITE [45]. In the *Waxman* scenario 1,250 nodes are randomly connected by a Waxman distribution. In *TopoAS* scenario 1,024 nodes are assigned to 32 autonomous systems which are connected by a backbone network. In all scenarios, a random link usage between 1 and 100 Mbit/s is assumed.

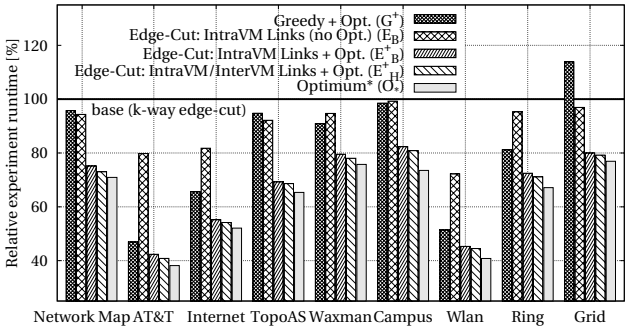

Figure 14. Performance of NETplace

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 is repeated 200 times.

Figure 14 shows the average experiment runtime of the placement strategies relative to a placement calculated by *minimized k-way edge-cut* using the *metis*[3] framework [28]. The figure shows the performance of the greedy approach with optimization (G^+), the balanced edge-cut approach without (E_B) and with optimization (E_B^+) and the hierarchical edge-cut approach (E_H^+). Due to the NP-hardness of the placement problem we cannot compute the optimum. To estimate the optimum (O^*), we computed for each scenario the best placement out of all runs with all algorithms.

Looking at the performance of G^+, it turns out that this simple greedy approach can outperform the reference algorithm in 8 of 9 scenarios. The consideration of the load introduced by intra-VM links (E_B) improves the placement in all scenarios. The additional optimization phase improves the runtime between 20% and 50%. The hierarchical approach E_H^+ can only slightly reduce the experiment runtime compared to E_B^+. As shown in the figure, the average experiment achieved with E_H^+ is almost as good as the minimal calculated runtime and compared to the reference algorithm we can reduce the runtime by up to 60%.

The time to calculate a placement using *NETplace* is contrary to the experiment reduction. Analysis of the scenarios with different numbers of virtual nodes and testbed sizes shows, that this time increases linearly with number of virtual nodes and increases sub-linearly with number of CPUs in the testbed. Additionally it turned out that the number of links in the topology as well as the topology itself has a large impact on the time to calculate the placement. During our evaluation this time is below 1min, except for the *Wlan* scenario along with testbed consisting of 128 and 256 CPUs where it takes up to 5min to calculate the placement. In contrast to the other scenarios the number of links per virtual node is very high resulting in a large number of neighboring states in the optimization phase.

[3]As proposed by the authors [46] of *metis*, we use *kmetis* to partition a graph into more than 8 partitions. Otherwise we use *pmetis*.

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

Table 3. Costs for virtual node migration

E. Dynamic Testbed Reconfiguration

We have implemented our approach by extending OpenVZ's checkpoint/restore functionality [33] for capturing frames which are queued in NETshaper during the migration of virtual nodes. 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 ran a set of micro benchmarks to identify the costs of migrations. Second, we emulate a scenario with 3 nodes and show that the migration does not biases the results. Third, using a synthetic evaluation based on this cost model and the testbed model, we evaluated the performance of NETbalance. The results of the micro benchmarks are summarized in Table 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 stopping 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.

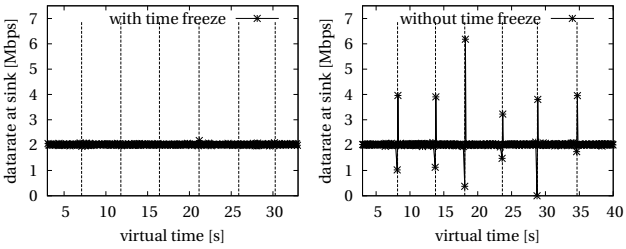

Figure 15. Emulation accuracy with migration

In order to show the emulation accuracy in presence of virtual node migrations, we emulated a scenario with 2 nodes connected to a third node (*sink*). The 2 nodes send a stream of UDP packets with 1Mbps 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 than one by one back to VM_1.

Figure 15 shows the data rate at the sink during the experiment. The right figure shows a migration of virtual

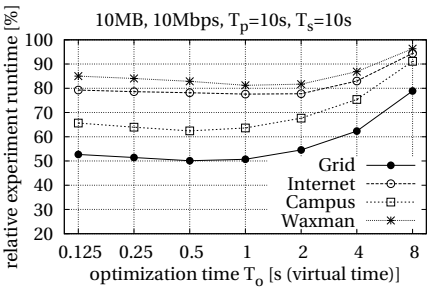

Figure 16. T_o vs. network topology

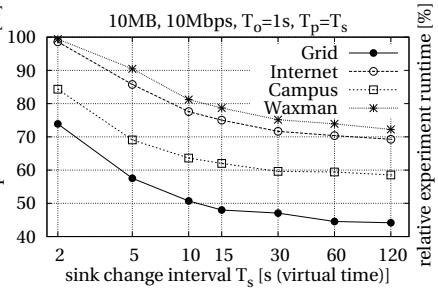

Figure 17. Migration benefit vs T_s

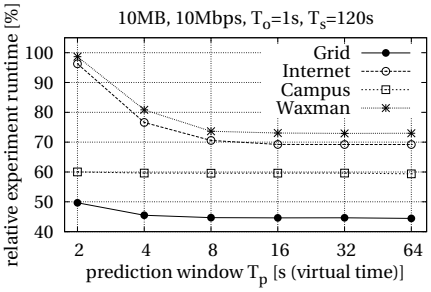

Figure 18. T_p vs. network topology

nodes without stopping the virtual time. Therefore, the a drop followed by a peak in the data rate is experienced by the sink. The left part shows the same scenario while stopping the virtual time during the migration. From the point of the software under test, the migration takes zero time, and the data rates are not influenced by the migration.

In the following, we show the potential of virtual node migration using 4 network topologies [10]: *Internet*, *Grid*, *Campus*, and *Waxman*. The used 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 10Mbps 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 pre-calculated. We use an initial placement optimized for the data flows to the first sink. If not otherwise stated, the sink changes every 2min. We use a testbed with 8 physical nodes each with 8 CPUs, and the *Grid* topology has a SuT allocating 10MB memory. The prediction window T_p is set to 10s, and the optimization time T_o is 1s.

In Figure 16, we present the influence of the optimization time T_o. In contrast to the other evaluations, here, we change the sink every 10s which is equal to the prediction window T_p. Even with a very short time of $T_o = 0.125s$, NETbalance can reduce the experiment runtime by up to 48%. Larger optimization times ($T_o \leq 1s$) 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. Increasing T_o further, 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 17 shows the influence of the sink change interval T_s on the experiment runtime. Here, we used a prediction window equal to the sink change interval and an optimization time of $T_o = 1s$. While increasing the sink change interval, the benefit of the migration increases.

Figure 18 shows the experiment runtime for prediction windows T_p between 1s and 64s for the different network topologies. For the *Campus* and the *Grid* scenario small values of T_p are enough for a significant speed-up. This mainly comes from the fact, that in these topologies small changes in the placement are enough to reduce the required τ significantly. At the same time, these small changes introduce only small reconfiguration costs which can be compensated even for short T_p. Regarding all topologies, a prediction window of 8s reduces the runtime between 27% and 55%.

VI. Conclusions

We have introduced the concepts of our *Network Emulation Testbed* (NET). NET provides highly scalable network emulation by combining *efficient node virtualization* and *adaptive virtual time*. This combination allows for running network experiments with thousands of unmodified instances of the software under test per testbed node. Using our network emulation tool *NETshaper* we can create an arbitrary network between these instances.

In order to minimize the runtime of these experiments, we introduced three extensions: First, we used *NETplace* to calculate an placement of the software instances onto the physical testbed nodes that maximizes the emulation speed. Second, *NETbalance* allows for adapting the placement during an experiment run to changed resource requirements of the software under test. Finally, we introduced *NETcaptain*, a management software to setup, control and visualize network experiments.

The combination of these concepts makes NET a versatile high-performance network emulation tool. In particular, NET is highly scalable, allowing thousands of virtual nodes per physical node due to its unique combination of node virtualization and adaptive virtual time.

References

[1] J. Liu, "Immersive Real-Time Large-Scale Network Simulation: A Research Summary," in *IPDPS'08*.
[2] R. M. Fujimoto, "Parallel Discrete Event Simulation," in *WSC'89*.
[3] G. F. Riley, "The Georgia Tech Network Simulator," in *MoMeTools'03*.
[4] B. Chun, D. Culler, T. Roscoe, A. Bavier, L. Peterson, M. Wawrzoniak, and M. Bowman, "PlanetLab: An Overlay Testbed for Broad-Coverage Services," *Comp. Comm. Rev.*, 2003.
[5] M. Hibler, R. Ricci, L. Stoller, J. Duerig, S. Guruprasad, T. Stack, K. Webb, and J. Lepreau, "Large-scale Virtualization in the Emulab Network Testbed," in *ATC'08*.
[6] A. Grau, S. Maier, K. Herrmann, and K. Rothermel, "Time Jails: A Hybrid Approach to Scalable Network Emulation," in *PADS'08*.

[7] G. Apostolopoulos and C. Chasapis, "V-eM: A Cluster of Virtual Machines for Robust, Detailed, and High-Performance Network Emulation," ICS-FORTH, Greece, Tech. Rep., 2006.
[8] D. Gupta, K. Yocum, M. McNett, A. C. Snoeren, A. Vahdat, and G. M. Voelker, "To Infinity and Beyond: Time-Warped Network Emulation," in *NSDI'06*.
[9] A. Grau, K. Herrmann, and K. Rothermel, "Efficient and Scalable Network Emulation Using Adaptive Virtual Time," in *ICCCN'09*.
[10] ——, "NETplace: Efficient Runtime Minimization of Network Emulation Experiments," in *SPECTS'10*.
[11] ——, "NETbalance: Reducing the Runtime of Network Emulation using Live Migration," in *ICCCN'11*.
[12] F. Bellard, "QEMU, a Fast and Portable Dynamic Translator," in *ATEC'05*.
[13] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield, "Xen and the Art of Virtualization," in *SOSP'03*.
[14] K. Kourai, T. Hirotsu, K. Sato, O. Akashi, K. Fukuda, T. Sugawara, and S. Chiba, "Secure and Manageable Virtual Private Networks for End-users," in *LCN'03*.
[15] P. H. Kamp and R. N. M. Watson, "Jails: Confining the omnipotent root," in *Sane'00*.
[16] E. Weingärtner, F. Schmidt, T. Heer, and K. Wehrle, "Synchronized Network Emulation: Matching prototypes with complex simulations," in *HotMetrics'08*.
[17] C. Bergstrom, S. Varadarajan, and G. Back, "The Distributed Open Network Emulator: Using Relativistic Time for Distributed Scalable Simulation," in *PADS'06*.
[18] M. Erazo, Y. Li, and J. Liu, "SVEET! A Scalable Virtualized Evaluation Environment for TCP," in *TridentCom'09*.
[19] M. D. Canon, D. H. Fritz, J. H. Howard, T. D. Howell, M. F. Mitoma, and J. Rodriquez-Rosell, "A Virtual Machine Emulator for Performance Evaluation," *Communications of the ACM*, 1980.
[20] G. F. Riley, R. M. Fujimoto, and M. H. Ammar, "A Generic Framework for Parallelization of Network Simulations," in *MASCOTS'99*.
[21] Shie-Yuan and H.-T. Kung, "A New Methodology for Easily Constructing Extensible and High-Fidelity TCP/IP Network Simulators," *Comp. Netw.*, 2002.
[22] R. Ricci, C. Alfeld, and J. Lepreau, "A Solver for the Network Testbed Mapping Problem," *Comp. Comm. Rev.*, 2003.
[23] Y. Liu, Y. Li, K. Xiao, and H. Cui, "Mapping Resources for Network Emulation with Heuristic and Genetic Algorithms," in *PDCAT'05*.
[24] J. Considine, J. W. Byers, and K. Meyer-Patel, "A Constraint Satisfaction Approach to Testbed Embedding Services," *Comp. Comm. Rev.*, 2004.
[25] P. Zheng and L. M. Ni, "EMPOWER: A Network Emulator for Wireless and Wireline Networks." in *INFOCOM'03*.
[26] B. Hendrickson and T. G. Kolda, "Graph partitioning models for parallel computing," *Parallel Computing*, 2000.
[27] X. Liu and A. A. Chien, "Realistic Large-Scale Online Network Simulation," in *SC'04*.
[28] G. Karypis and V. Kumar, "A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs," *SIAM Journal on Scientific Computing*, 1998.
[29] N. Shivaratri, P. Krueger, and M. Singhal, "Load Distributing for Locally Distributed Systems," *Computer*, 2002.
[30] M. Willebeek-LeMair and A. Reeves, "Strategies for dynamic load balancing on highly parallel computers," *Transactions on Parallel and Distributed Systems*, 2002.
[31] M. M. Theimer, K. A. Lantz, and D. R. Cheriton, "Preemptable Remote Execution Facilities for the V-System," in *SOSP'85*.
[32] E. Zayas, "Attacking the Process Migration Bottleneck," in *SOSP'87*.
[33] OpenVZ, http://openvz.org, 2011.
[34] D. Herrscher and K. Rothermel, "A Dynamic Network Scenario Emulation Tool," in *ICCCN'02*.
[35] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield, "Live Migration of Virtual Machines," in *NSDI'05*.
[36] P. A. Dinda and D. R. O'Hallaron, "Host load prediction using linear models," *Cluster Computing*, 2000.
[37] L. Yang, I. Foster, and J. M. Schopf, "Homeostatic and Tendency-based CPU Load Predictions," in *IPDPS'03*.
[38] S. Osman, D. Subhraveti, G. Su, and J. Nieh, "The Design and Implementation of Zap: A System for Migrating Computing Environments," *Op. Sys. Rev.*, 2002.
[39] M. Kozuch and M. Satyanarayanan, "Internet Suspend/Resume," in *WMCSA'02*.
[40] netperf, http://netperf.org, 2011.
[41] T. McGregor, H.-W. Braun, and J. Brown, "The nlamr network analysis infrastructure," *Comm. Magazine*, 2000.
[42] Rocketfuel, "PoP-level ISP maps (policy-dist.tar.gz)," Data file available at http://cs.washington.edu/research/ networking/rocketfuel/, 2003.
[43] ——, "ISP maps (rocketfuel_maps_cch.tar.gz)," Data file available at http://cs.washington.edu/research/networking/ rocketfuel/, 2003.
[44] C. Kiddle, R. Simmonds, and B. Unger, "Improving Scalability of Network Emulation through Parallelism and Abstraction," in *ANSS'05*.
[45] A. Medina, A. Lakhina, I. Matta, and J. Byers, "BRITE: An Approach to Universal Topology Generation," in *MASCOTS'01*.
[46] G. Karypis and V. Kumar, "METIS, a Software Package for Partitioning Unstructured Graphs and Computing Fill-Reduced Orderings of Sparse Matrices," University of Minnesota, Tech. Rep., 1998.

Andreas Grau is a Ph.D. candidate at the Distributed Systems research group at the University of Stuttgart, Germany. His research interests include scalability of network emulation. Grau holds a Dipl.-Inf. (MS) degree in computer science from the University of Stuttgart.

Klaus Herrmann studied Computer Science at the University of Frankfurt/Main, Germany. He received his Ph.D. from Berlin University of Technology. Since September 2006 he holds a postdoctoral research position at the University of Stuttgart. His research interests are distributed systems, mobile computing, and self-organizing software systems.

Kurt Rothermel is a professor in the Distributed Systems research group at the University of Stuttgart. His research interests include performance evaluation of distributed systems, context aware and adaptive systems, and sensor networks. Rothermel received a PhD in computer science from the University of Stuttgart. He is a member of the ACM and the GI.