NETbalance: Reducing the Runtime of Network Emulation using Live Migration

Andreas Grau, Klaus Herrmann, and Kurt Rothermel
Universität Stuttgart, Institute of Parallel and Distributed Systems (IPVS)
Universitätsstr. 38, D–70569 Stuttgart, Germany
Email: {grau,herrmann,rothermel}@ipvs.uni-stuttgart.de
Telephone: +49 (711) 685–88236, Fax: –88424

***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 instances of the software under test (called *virtual nodes*) in a controlled environment. In previous work, we introduced an approach to minimize the runtime of network emulation experiments based on prior known average resource requirements of virtual nodes.**

In this paper, we introduce *NETbalance*, a novel approach to runtime reduction for experiments with *unknown or varying resource requirements*. NETbalance migrates virtual nodes during an experiment to distribute the load evenly across the physical nodes, avoiding overloaded nodes and exploiting the idle resources on underloaded nodes for speeding up the experiment execution. We make the following contributions: First, we present an emulation architecture for efficiently supporting live migration of virtual nodes. Second, we propose a cost model for determining the runtime reduction achieved through the migration. Third, we introduce an algorithm for calculating placements that minimize the experiment runtime. Our evaluations of the NETbalance prototype show, that it is able to reduce the experiment runtime by up to 70%.

***Index Terms*—network emulation; virtual time; live migration; virtual node placement**

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 [14], [6], [17], real-world testbeds [3], and network emulation [11], [9], [1]. 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). 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 [7], 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, this slowdown increases the runtime of an experiment, which should be minimal for maximum testbed efficiency. Since dedicated hardware (physical nodes and networks) is typically used for network experiments, cost factors like bandwidth or CPU usage are not an issue. Therefore, our goal is to minimize the runtime of the experiments on the given hardware.

In order to reach this goal, we have developed NETplace [8]. The basic idea is to calculate an initial placement of virtual nodes onto physical nodes that minimizes the load of the physical nodes and, therefore, minimizes the experiment runtime. The NETplace algorithm assumes prior knowledge of the average experiment load including the network and CPU usage of virtual nodes. However, in experiments with varying load this approach may lead to temporary suboptimal placements and, thus, longer experiment runtimes.

In this paper, we present *NETbalance*, a extended approach to minimizing the experiment runtime in scenarios with varying and unknown load. *NETbalance* monitors the load of the virtual nodes to detect load changes and trigger the migration of virtual nodes during the 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 cause of suboptimal experiment runtimes. A new placement is only deployed if the resulting speed-up outweighs the cost for the migrations. The contributions of this paper are as follows:

1) an *emulation architecture* to efficiently support migration of virtual nodes,
2) a *migration cost model* to determine the time required for the migration of virtual nodes,
3) an *algorithm* to calculate a re-placement of virtual nodes, which reduces the experiment runtime,

We built a prototype of NETbalance in our network emulator. Our extensive evaluations show that NETbalance can reduce the experiment runtime up to 70%.

The remainder of this paper is structured as follows. In Section II, we introduce the system architecture of our network emulator. The approach to calculate an optimized virtual node

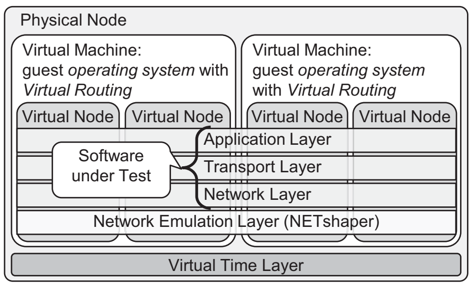
Physical Node
Virtual Machine: guest *operating system* with *Virtual Routing*
Virtual Machine: guest *operating system* with *Virtual Routing*
Virtual Node
Virtual Node
Virtual Node
Virtual Node
Software under Test
Application Layer
Transport Layer
Network Layer
Network Emulation Layer (NETshaper)
Virtual Time Layer

Fig. 1. TVEE Architecture

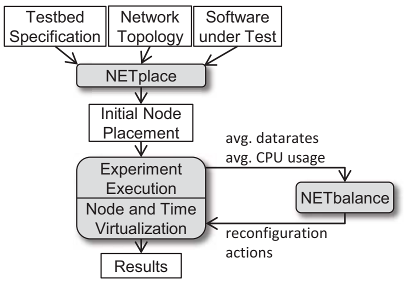

Fig. 2. Experiment execution workflow

placement is discussed in Section III. In Section IV, we extend our emulation architecture to efficiently support the migration of virtual nodes and introduce a cost model to calculate the resulting reconfiguration costs. The detailed evaluation in Section V emphasizes the effectiveness and the efficiency of NETbalance. In Section VI, we present related work before we conclude the paper with a summary and a discussion of future work in Section VII.

II. System Architecture

In order to support highly scalable network experiments in NET, we have developed the *Time Virtualized Emulation Environment* (TVEE) [9]. As shown in Figure 1, TVEE is based on two building blocks: *node virtualization* and *time virtualization*. Node virtualization [9] allows for executing multiple virtual nodes running the *Software under Test* (SuT) on a single physical node. Time virtualization [7] 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 [7] 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.

In order to provide virtual time transparently to the SuT, we make use of the virtual machine (VM) abstraction [2]. For maximum efficiency, we run one VM on each CPU of a physical node [8]. *Virtual Routing* [12], [15] allows for creating virtual nodes [9] 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* [10], 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.

III. System Reconfiguration

Running an experiment using NET follows the workflow depicted in Figure 2. 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 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 extends this workflow to minimize the experiment runtime in scenarios with varying resource requirements. At runtime, NETbalance detects changes in the resource requirements of virtual nodes. These changes trigger the recalculation of the virtual nodes' placement. We adopt the concept of *live migration* [4] 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 [5]. As reported by Yang et. al. [21] 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* [8] 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 (cf. Section IV) 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:

$$\chi = [(T_p - T_o) \cdot \tau_{\phi'} + T_r] - (T_p - T_o) \cdot \tau_\phi \qquad (1)$$

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 reconfiguration with minimal gain.

IV. Live Migration of Virtual Nodes

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.

[1]Scenario with 4 network links per virtual node

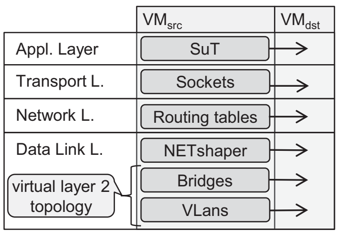

Fig. 3. Migration of virtual node's memory state

After the experiment is stopped, we change the placement of virtual nodes. For this, we adopt the concepts of the ZAP system [16]. First, we create a snapshot of a virtual node using check-pointing. Figure 3 shows the state of a virtual node for each 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. The state of the virtual node is then transferred to VM$_{\text{dst}}$ 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$_{\text{dst}}$. 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] (see Figure 4). 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 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

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

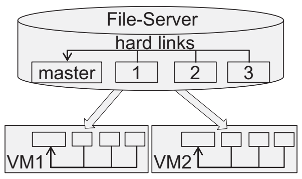

Fig. 4. File systems of the virtual nodes

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 1. Costs for virtual node migration

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 [13], the reconfiguration time T_r is defined as follows:

$$T_r = T_{\text{suspend}} + T_{\text{migrate}} + T_{\text{change-topology}} + T_{\text{resume}} \qquad (2)$$

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.

V. Evaluation

We have implemented our approach by extending OpenVZ's checkpoint/restore functionality [15] 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 two steps. First, we ran a set of micro benchmarks to identify the costs of migrations. Second, using a synthetic evaluation based on this cost model, we evaluated the performance of NETbalance. The results of the micro benchmarks are summarized in Table 1. 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.

We evaluated NETbalance using four network topologies [8]: (1) an *Internet* scenario based on a snapshot of the Internet with 2,113 routers, (2) a *Grid* model with 1,600 nodes arranged in a regular square grid, (3) a *Campus* model with 5,480 nodes modeling connected campus sites, and (4) a *Waxman* model with 2,500 nodes connected according to a Waxman distribution.

In order to show the potential of NETbalance, we emulate 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.

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 prediction window T_p, the optimization time T_o, 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 shows the required time dilation factor τ for running the scenario with a Grid topology. The gray area shows τ 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 τ. The black line in the figure shows τ in the same scenario using NETbalance. As soon as the sink changes, τ rises. However, after the deployment of an optimized placement, τ goes back to the original level. The different values of τ are caused by the location of the sink in the network.

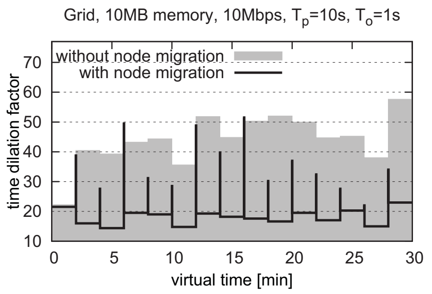

Fig. 5. Experiment speed over time

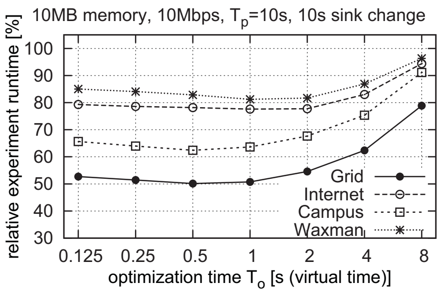

Fig. 6. Optimization time vs. network topology

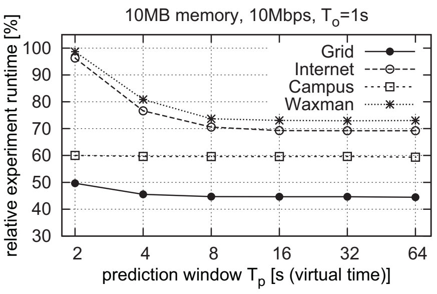

Fig. 7. Prediction window vs. network topology

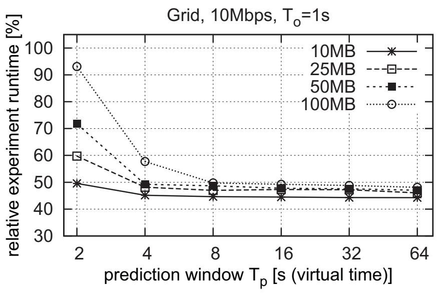

Fig. 8. Prediction window vs. memory of the SuT

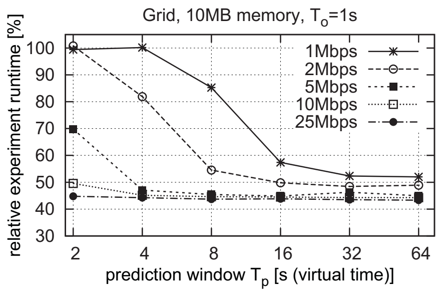

Fig. 9. Prediction window vs. data rate

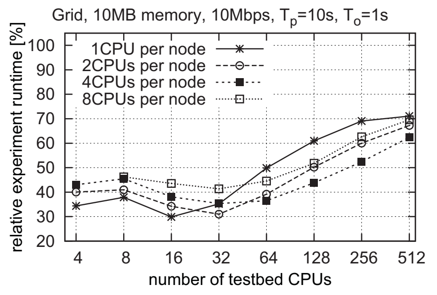

Fig. 10. Migration benefit vs. testbed size

In Figure 6, 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 7 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%.

Since the memory footprint of the SuT determines the reconfiguration costs, we evaluated the required prediction window for different footprint sizes. Figure 8 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 a 100MB of used memory pages, a prediction window of $T_p \geq 8s$ is sufficient for reducing the experiment runtime.

In contrast to the memory footprint, higher data rates between virtual nodes are beneficial for NETbalance (see Figure 9). Higher data rates result in higher load of the physical nodes and, therefore, a higher time dilation factor. In contrast, the reconfiguration costs 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 (see Figure 10). 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 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 a 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.

VI. Related Work

Up to now, there has been no network emulator that uses the migration of virtual nodes to reduce the runtime of experiments. 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 in parallel computing. Here, load balancing [18], [20] is achieved by migrating tasks from nodes with high load to nodes with low load. Willebeek-LeMair and Reeves [20] 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, here, 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* [19] and *demand-migration* [22] minimizes 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 transfered 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.

Therefore, none of the existing approaches can minimize the runtime of experiments with unknown or varying load.

VII. Conclusion

In this paper, we introduced a novel emulation approach, called *NETbalance*, for reducing the runtime of experiments with unknown or dynamic resource requirements. The basic idea of *NETbalance* is to migrate virtual nodes during the experiment runtime to minimize the load of the maximum loaded physical nodes and, thus, to minimize runtime of the entire experiment. We presented an emulation architecture to efficiently support live migration of virtual nodes. In order to calculate the experiment runtime reduction achieved through the migration, we proposed a migration cost model. Based on the cost model, we developed an algorithm for calculating a placement that minimizes the remaining experiment runtime.

We implemented *NETbalance*, and our evaluation of this prototype shows that live migration can reduce the runtime of network experiments by up to 70% for various network topologies and load characteristics.

This represents a major improvement of emulation technology. The *NETbalance* concepts enable scientists to run more experiments in less time, achieving statistically more relevant results. Moreover, the effort of preparing experiments is drastically decreased as prior knowledge about application behavior is no longer needed.

In our future work, we will investigate means for further reducing experiment runtime by creating snapshots of the complete experiment. Setting up an experiment from such a snapshot speeds up evaluations that require long initial setup phases.

VIII. Acknowledgment

This work is partially funded by the Deutsche Forschungsgemeinschaft (German Research Foundation) under grant DFG-GZ RO 1086/9-3. We thank Sebastian Bartmann for helping with the implementation of the *NETbalance* prototype.

References

[1] G. Apostolopoulos and C. Chasapis. V-eM: A Cluster of Virtual Machines for Robust, Detailed, and High-Performance Network Emulation. Technical Report 371, 2006.

[2] 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 *Proceeding of the 19th ACM Symposium on Operating Systems Principles (SOSP'03)*, 2003.

[3] B. Chun, D. Culler, T. Roscoe, A. Bavier, L. Peterson, M. Wawrzoniak, and M. Bowman. PlanetLab: An Overlay Testbed for Broad-Coverage Services. *ACM SIGCOMM Computer Communication Review*, 33(3), 2003.

[4] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live Migration of Virtual Machines. In *Proceedings of the 2nd Symposium on Networked Systems Design & Implementation (NSDI'05)*, 2005.

[5] P. A. Dinda and D. R. O'Hallaron. Host load prediction using linear models. *Cluster Computing*, 3, 2000.

[6] R. M. Fujimoto. Parallel Discrete Event Simulation. In *Proceedings of the 21st conf. on Winter simulation (WSC'89)*, 1989.

[7] A. Grau, K. Herrmann, and K. Rothermel. Efficient and Scalable Network Emulation Using Adaptive Virtual Time. In *Proceedings of 18th International Conference on Computer Communications and Networks (ICCCN'09)*, 2009.

[8] A. Grau, K. Herrmann, and K. Rothermel. NETplace: Efficient Runtime Minimization of Network Emulation Experiments. In *Proceedings of the International Symposium on Performance Evaluation of Computer and Telecomm. Systems (SPECTS'10)*, 2010.

[9] A. Grau, S. Maier, K. Herrmann, and K. Rothermel. Time Jails: A Hybrid Approach to Scalable Network Emulation. In *22nd ACM/IEEE/SCS Workshop on Principles of Advanced and Distributed Simulations (PADS'08)*, 2008.

[10] D. Herrscher and K. Rothermel. A Dynamic Network Scenario Emulation Tool. In *Proceedings of the 11th International Conference on Computer Communications and Networks (ICCCN'02)*, 2002.

[11] 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 *USENIX 2008 Annual Technical Conference on Annual Technical Conference (ATC'08)*, 2008.

[12] 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 *Proceedings of the 28th Annual IEEE International Conference on Local Computer Networks (LCN'03)*, 2003.

[13] M. Kozuch and M. Satyanarayanan. Internet Suspend/Resume. In *Proceedings of the Fourth IEEE Workshop on Mobile Computing Systems and Applications (WMCSA'02)*, Washington, DC, USA, 2002.

[14] J. Liu. Immersive Real-Time Large-Scale Network Simulation: A Research Summary. *Proceedings of the IEEE International Symposium on Parallel and Distributed Processing (IPDPS'08)*, 2008.

[15] OpenVZ. http://openvz.org, 2011.

[16] S. Osman, D. Subhraveti, G. Su, and J. Nieh. The Design and Implementation of Zap: A System for Migrating Computing Environments. *SIGOPS Operating System Review*, 36, 2002.

[17] G. F. Riley. The Georgia Tech Network Simulator. In *Proceedings of the ACM SIGCOMM workshop on Models, methods and tools for reproducible network research (MoMeTools'03)*. ACM, 2003.

[18] N. Shivaratri, P. Krueger, and M. Singhal. Load Distributing for Locally Distributed Systems. *Computer*, 25(12), 2002.

[19] M. M. Theimer, K. A. Lantz, and D. R. Cheriton. Preemptable Remote Execution Facilities for the V-System. In *Proceedings of the tenth ACM symposium on Operating systems principles (SOSP'85)*, 1985.

[20] M. Willebeek-LeMair and A. Reeves. Strategies for dynamic load balancing on highly parallel computers. *IEEE Transactions on Parallel and Distributed Systems*, 4(9), 2002.

[21] L. Yang, I. Foster, and J. M. Schopf. Homeostatic and tendency-based CPU load predictions. In *Proceedings of the International Parallel and Distributed Processing Symposium (IPDPS'03)*, 2003.

[22] E. Zayas. Attacking the Process Migration Bottleneck. In *Proceedings of the eleventh ACM Symposium on Operating systems principles (SOSP '87)*, 1987.