NORTHWESTERN
UNIVERSITY

Computer Science Department

Technical Report
NWU-CS-05-13
June 28, 2005

Free Network Measurement For Adaptive Virtualized Distributed Computing

Ashish Gupta Marcia Zangrilli[*]
Ananth Sundararaj Peter A. Dinda Bruce B. Lowekamp[*]

Abstract

An execution environment consisting of virtual machines (VMs) interconnected with a virtual overlay network can use the naturally occurring traffic of an existing, unmodified application running in the VMs to measure the underlying physical network. Based on these characterizations, and characterizations of the application's own communication topology, the execution environment can optimize the execution of the application using application-independent means such as VM migration and overlay topology changes. In this paper, we demonstrate the feasibility of such free automatic network measurement by fusing the Wren passive monitoring and analysis system with Virtuoso's virtual networking system. We explain how Wren has been extended to support on-line analysis, and we explain how Virtuoso's adaptation algorithms have been enhanced to use Wren's physical network level information to choose VM-to-host mappings, overlay topology, and forwarding rules.

Effort sponsored by the National Science Foundation under Grants ANI-0093221, ACI-0112891, ANI-0301108, ACI-0203974, EIA-0130869, EIA-0224449, and gifts from VMware, Sun, and Dell. This work was performed in part using computational facilities at the College of William and Mary that were enabled by grants from Sun Microsystems, the National Science Foundation, and Virginia's Commonwealth Technology Research Fund. Zangrilli is partially supported by an Aerospace Graduate Research Fellowship from the Virginia Space Grant Consortium. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation (NSF), Virginia, VMware, Sun, or Dell.

[*] Department of Computer Science, College of William and Mary

Keywords: virtual machines, adaptive systems, network measurement, overlay networks

Free Network Measurement For Adaptive Virtualized Distributed Computing

Ashish Gupta Marcia Zangrilli Ananth Sundararaj
Peter Dinda Bruce B. Lowekamp

Abstract

An execution environment consisting of virtual machines (VMs) interconnected with a virtual overlay network can use the naturally occurring traffic of an existing, unmodified application running in the VMs to measure the underlying physical network. Based on these characterizations, and characterizations of the application's own communication topology, the execution environment can optimize the execution of the application using application-independent means such as VM migration and overlay topology changes. In this paper, we demonstrate the feasibility of such free automatic network measurement by fusing the Wren passive monitoring and analysis system with Virtuoso's virtual networking system. We explain how Wren has been extended to support on-line analysis, and we explain how Virtuoso's adaptation algorithms have been enhanced to use Wren's physical network level information to choose VM-to-host mappings, overlay topology, and forwarding rules.

1 Introduction

Virtual machines interconnected with virtual networks are an extremely effective platform for high performance distributed computing, providing benefits of simplicity and flexibility to both users and providers [1, 4]. We have developed a virtual machine distributed computing system called Virtuoso [13] that is based on virtual machine monitors[1] and a virtual overlay network system called VNET [14].

A platform like Virtuoso also provides key opportunities for resource and application monitoring, and adaptation. In particular, it can:

1. Monitor the application's traffic to automatically and cheaply produce a view of the application's network demands. We have developed a tool, VTTIF [2], that accomplishes this.
2. Monitor the performance of the underlying physical network by use the application's own traffic to automatically and cheaply probe it, and then use the probes to produce characterizations. This paper describes how this is done.
3. Adapt the application to the network to make it run faster or more cost-effectively. This paper extends our previous adaptation work [16, 15] with algorithms that make use of network performance information.
4. Reserve resources, when possible, to improve performance [6, 7].

Virtuoso is capable of accomplishing these feats using existing, unmodified applications running on existing, unmodified operating systems.

We build on the success of our Wren passive monitoring and network characterization system [18, 17] to accomplish (2) above. Wren consists of a kernel extension and a user-level daemon. Wren can:

1. Observe every incoming and outgoing packet arrivals in the system with low overhead.
2. Analyze these arrivals using state-of-the-art techniques to derive from them latency and bandwidth information for all hosts that the present host communications with. Earlier work described offline analysis techniques. This paper describes online techniques to continuously and dynamically update the host's view of the network.
3. Collect latency, available bandwidth, and throughput information so that an adaptation algorithm can have a bird's eye view of the physical network, just as it has a bird's eye view of the application topology via VTTIF. This new work is described for the first time here.

Effort sponsored by the National Science Foundation under Grants ANI-0093221, ACI-0112891, ANI-0301108, ACI-0203974, EIA-0130869, EIA-0224449, and gifts from VMware, Sun, and Dell. This work was performed in part using computational facilities at the College of William and Mary that were enabled by grants from Sun Microsystems, the National Science Foundation, and Virginia's Commonwealth Technology Research Fund. Zangrilli is partially supported by an Aerospace Graduate Research Fellowship from the Virginia Space Grant Consortium. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation (NSF), Virginia, VMware, Sun, or Dell.

[1] Specifically, VMWare GSX Server 2.5, although that is not a requirement of either Virtuoso nor of the work described in this paper. The VNET virtual networking component of Virtuoso requires only that a virtual interface be exposed. It has been used successfully with entirely different VMMs, such as User Mode Linux. Wren's kernel extensions and userland analysis daemon require only Linux.

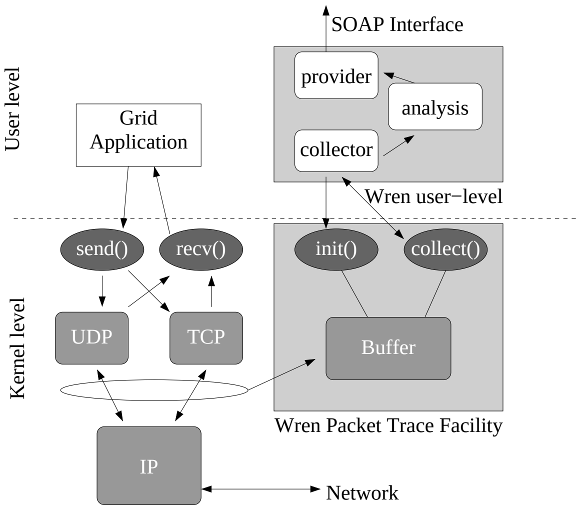


Figure 1. Wren architecture.

4. Answer queries about the bandwidth and latency between any pair of machines in the virtual network. This is described for the first time here.

In the following, we begin by describing and evaluating the online Wren system (Section 2) and how it interacts with the Virtuoso system (Section 3). In Section 4, we describe adaptation algorithms in Virtuoso that make use of Wren's view of the physical network. While we evaluate Wren in a live environment and use live performance data, our evaluation of the adaptation algorithms is currently limited to simulation.

2 Wren online

The Wren architecture is shown in Figure 1. The key feature Wren uses is kernel-level packet trace collection. These traces allow precise timestamps of the arrival and departure of packets on the machines. The precision of the timestamps is necessary to observe the behavior of small groups of packets on the network. A user-level component collects the traces from the kernel. Run-time analysis determines available bandwidth and the measurements are reported to other applications through a SOAP interface. Alternatively, the packet traces can be transmitted to a remote repository.

Because we are targeting applications with potentially bursty and irregular communication patterns, many applications will not generate enough traffic to saturate the network and provide useful information on the current bandwidth achievable on the network. The key observation behind Wren is that even when the application is not saturating the network it is sending bursts of traffic that can be used to measure the available bandwidth of the network.

2.1 Online analysis

Wren's general approach and collection overhead have been presented and analyzed in previous papers [18, 17]. To support Virtuoso's adaptation, however, two changes are required. First, previous implementations of Wren have relied on static offline analysis. We describe here our online analysis algorithm used to report available bandwidth measurements using our SOAP interface. Second, previous implementations have relied on detecting fixed-size bursts of network traffic. The new online tool scans for maximum-sized trains that can be formed using the collected traffic. This approach results in more measurements taken from a smaller amount of traffic.

The analysis used by WrenTool is based on the self-induced congestion (SIC) algorithm [10, 11]. Active implementations of this algorithm generate trains of packets at progressively faster rates until increases in one-way delay are observed, indicating queues building along the path resulting from the available bandwidth being consumed. We apply similar analysis to our passively collected traces, but our key challenge is identifying appropriate trains from the stream of packets generated by the TCP sending algorithm. ImTCP integrates an active SIC algorithm into a TCP stack, waiting until the congestion window has opened large enough to send an appropriate length train and then delaying packet transmissions until enough packets are queued to generate a precisely spaced train [8]. Wren avoids modifying the TCP sending algorithm, and in particular delaying packet transmission.

The tradeoff between the two types of active implementation is that Wren must select the data naturally available in the TCP flow. Although Wren has less control over the trains and selects shorter trains than would deliberately be generated by active probing, over time the burstiness of the TCP process produces many trains at a variety of rates [12], thus allowing bandwidth measurements to be made. There are elements in common with TCP Vegas, Westwood, and FastTCP, but those approaches deliberately increase the congestion window until one-way delay increases, whereas we do not require the congestion window to expand until long-term congestion is observed and can detect congestion using bursts at slower average sending rates.

The online WrenTool sorts information about incoming and outgoing packets by source and destination IP address. For each source/destination pair we sort the packets by increasing sequence number and apply our one-sided available bandwidth algorithm.

The first step in our one-sided algorithm is to group packets into trains. We look at the relationship between the interdeparture times of sequential data packets. If interdeparture times of successive pairs are similar, then the packets are departing the machine at approximately the same rate. Let Δ_0 be the interdeparture time between data packets 0 and 1. To form a train, the interdeparture times Δ_i between each successive pair of packets i and $i+1$ in the train must satisfy the requirement that $\min_i(\log(\Delta_i)) > \max_j(\log(\Delta_j)) - \alpha$, essentially requiring consistent spacing between the packets. For these experiments, we accepted trains where $\alpha = 1$. Because of the bursty transmission of packets within any TCP flow [12], interdeparture times typically vary by several orders of magnitude even during bulk data transfers, therefore this approach selects only the more consistently spaced bursts as valid trains. We impose a minimum length of 7 packets for valid trains.

The first step in processing a train is to use a pairwise comparison test to determine the trend in the RTTs of the packets in that train. If $\forall i : RTT_i < RTT_{i+1}$ then the train has an increasing trend. Otherwise, the train trend is labeled as non-increasing. Next, we calculate the initial sending rate (ISR) by dividing the total number of bits in the train by the difference between the end time and the start time. If the train has a non increasing trend, we know that the train ISR did not cause queuing on the path. Therefore, we report the ISR as our lower bound available bandwidth measurement when the trend of the train is non increasing. Conversely, if a train presents an increasing trend, its ISR is an upper bound for the available bandwidth.

All available bandwidth observations are passed to the Wren observation thread. The observation thread provides a SOAP interface that clients can use to receive the stream of measurements produced using application traffic. Because the trains are short and represent only a singleton observation of an inherently bursty process, multiple observations are required to converge to an accurate measurement of available bandwidth.

2.2 Overheads

Our evaluation of the overhead of using Wren for analysis has shown that the collection of data in the kernel adds essentially no overhead to data transmission and reception, and has no effect on throughput or latency. The primary sources of overhead are copying the trace data to user-level and either the cost of local analysis or of sending the data to another machine to be analyzed. Figure 2 shows the runtime of the STREAM benchmark across a range of polling rates. Here the data is collected from the kernel and transmitted to a remote host for analysis at the given intervals. Because the STREAM benchmark does not saturate the host's network connection, the outgoing packet traces have no effect. We have found that polling intervals of one second are sufficient to capture an application's traffic without influencing that application's performance.

2.3 Performance

First, we evaluate our new variable train-length algorithm against controlled congestion on a closed testbed. We generate a fixed-rate UDP stream that shares the same link as the traffic between the machines. We used iperf to generate TCP traffic on this link, reporting its throughput at 1 second intervals. Figure 3 illustrates the resulting observations. The WrenTool is able to measure the available bandwidth on the link throughout the experiment regardless of whether iperf is currently saturating the link.

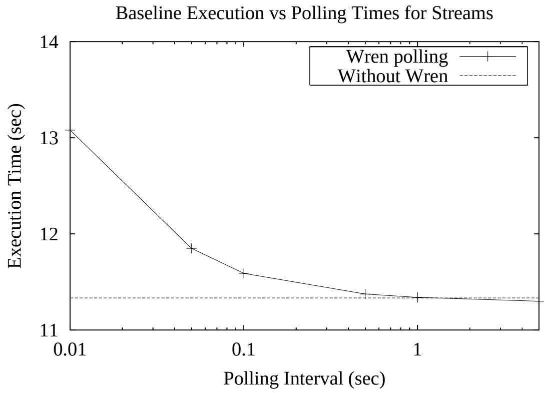


Figure 2. The execution time of the STREAM benchmark when monitored by the Wren system using a range of polling intervals.

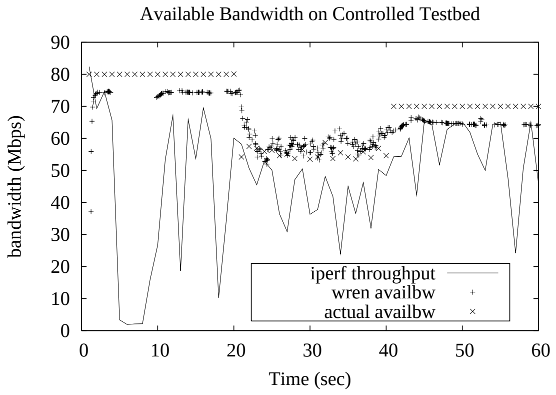


Figure 3. The results of applying the WrenTool to bursty traffic generated on a testbed with a controlled level of available bandwidth.

To validate the combination of Wren monitoring an application using VNET we ran a simple BSP-style communication pattern generator. Figure 4 shows the results of this experiment, with the throughput achieved by the application during its bursty communication phase and Wren's available bandwidth observations. Although the application never achieved significant levels of throughput, Wren was able to measure the available bandwidth. Validating these results across a WAN is difficult, but iperf achieved approximately 24Mbps throughput when run following this experiment, which is in line with our expectations based on Wren's observations and the large number of connections sharing W&M's 150Mbps Abilene connection.

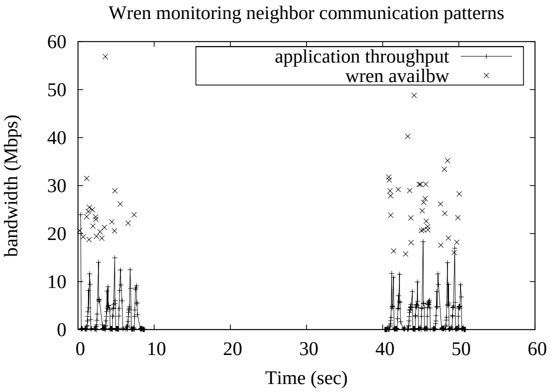


Figure 4. Wren observing a neighbor communication pattern sending 200K messages within VNET.

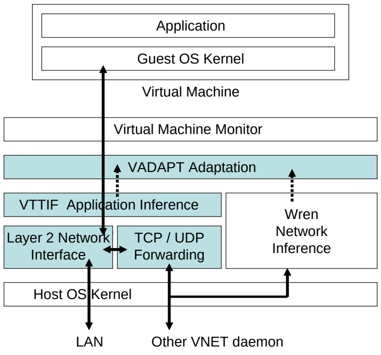


Figure 5. Virtuoso's interaction with Wren. The highlighted boxes are components of Virtuoso.

3 Virtuoso and Wren

Virtuoso [13, 1], is a system for virtual machine distributed computing where the virtual machines are interconnected with VNET, a virtual overlay network. The VTTIF (virtual traffic and topology inference framework) component observes every packet sent by a VM and infers from this traffic a global communication topology and traffic load matrix among a collection of VMs. Wren uses the traffic generated by VNET to monitor the underlying network and makes its measurements available to Virtuoso's adaptation framework, as seen in Figure 5.

3.1 VNET

VNET [14, 16] is the part of Virtuoso that creates and maintains the networking illusion that the user's virtual machines (VMs) are on the user's local area network. Each physical machine that can instantiate virtual machines (a host) runs a single VNET daemon. One machine on the user's network also runs a VNET daemon. This machine is referred to as the Proxy. Each of the VNET daemons is connected by a TCP or a virtual UDP connection (a VNET link) to the VNET daemon running on the Proxy. This is the initial star topology that is always maintained. Additional links and forwarding rules can be added or removed at any time to improve application performance.

The VNET daemon running on a machine opens the machine's virtual (i.e., VMM-provided attachments to the VMs' interfaces) and physical Ethernet interfaces in promiscuous mode. Each packet captured from an interface or received on a link is matched against a forwarding table to determine where to send it, the possible choices being sending it over one of the daemon's outgoing links or writing it out to one of the local interfaces. Each successfully matched packet is also passed to VTTIF to determine the local traffic matrix. Each VNET daemon periodically sends its inferred local traffic matrix to the VNET daemon on the Proxy. The Proxy, through its physical interface, provides a network presence for all the VMs on the user's LAN and makes their configuration a responsibility of the user and his site administrator.

3.2 VTTIF

The VTTIF component integrates with VNET to automatically infer the dynamic topology and traffic load of applications running inside the VMs in the Virtuoso system. In our earlier work [2], we demonstrated that it is possible to successfully infer the behavior of a BSP application by observing the low level traffic sent and received by each VM in which it is running. We have also shown [16] how to smooth VTTIF's reactions so that adaptation decisions made on its output cannot lead to oscillation. The reaction time of VTTIF depends on the rate of updates from the individual VNET daemons and on configuration parameters. Beyond this rate, we have designed VTTIF to stop reacting, settling into a topology that is a union of all the topologies that are unfolding in the network.

VTTIF works by examining each Ethernet packet that a VNET daemon receives from a local VM. VNET daemons collectively aggregate this information producing a global traffic matrix for all the VMs in the system. To provide a stable view of dynamic changes, it applies a low pass filter to the updates, aggregating the updates over a sliding window and basing its decisions upon this aggregated view. The application topology is then recovered from this matrix by applying normalization and pruning techniques.

Since the monitoring is done below the VM, it does not depend on the application or the operating system in any manner. VTTIF automatically reacts to interesting changes in traffic patterns and reports them, driving adaptation.

3.3 Integrating Virtuoso and Wren

Virtuoso and Wren are integrated by incorporating the Wren extensions into the Host operating system of the machines running VNET. In this position, Wren monitors the traffic between VNET daemons, not between individual VMs. Both the VMs and VNET are oblivious to this monitoring, except for a small performance degradation.

The local instance of Wren is made visible to Virtuoso through it's SOAP interface. VTTIF executes nonblocking calls to Wren to collect updates on available bandwidth and latency from the local host to other VNET hosts. VTTIF uses VNET to periodically send the local matrices to the Proxy machine, which maintains global matrices with information about every pair of VNET hosts. In practice, only those pairs whose VNET daemons exchange messages have entries. Through these mechanisms, the Proxy has a view of the physical network interconnecting the machines running VNET daemons and a view of the application topology and traffic load of the VMs.

3.4 Overheads

The overheads of integrating Wren with Virtuoso stem from the extra kernel-level Wren processing each VNET transmission sees, Wren user-level processing of data into bandwidth and latency estimates, and the cost of using VNET and VTTIF to aggregate local Wren information into a global view. Of these, only the first is in the critical path of application performance. As described in Section 2.2, the kernel-level processing has no distinguishable effect on either throughput or latency. With VTTIF, latency is unaffected, while throughput is affected by ∼1%. The cost of local processing is tiny and can be delayed.

4 Adaptation using network information

As shown in Figure 5, the VADAPT component of Virtuoso, using the VTTIF and Wren mechanisms, has a view of the dynamic performance characteristics of the physical network interconnecting the machines running VNET daemons and a view of the the demands that the VMs place on it. More specifically, it receives:

1. A graph representing the application topology of the VMs and a traffic load matrix among them, and
2. Matrices representing the available bandwidth and latency among the Hosts running VNET daemons.

VADAPT's goal is to use this information to choose a configuration that maximizes the performance of the application running inside the VMs. A configuration consists of

1. The mapping of VMs to Hosts running VNET daemons,
2. The topology of the VNET overlay network,
3. The forwarding rules on that topology, and
4. The choice of resource reservations on the network and the hosts, if available.

In previous work [16, 15], we have demonstrated heuristic solutions to a subset of the above problem. In particular, we have manipulated the configuration (sans reservations) in response to application information. In the following, we expand this work in two ways. First, we show how to incorporate the information about the physical network in a formal manner. Second, we describe two heuristic approaches for addressing the formal problem and present an initial evaluation of them.

4.1 Problem formulation

VNET Topology: We are given a complete directed graph $G = (H, E)$ in which H is the set of all of the Hosts that are running VNET daemons and are capable of hosting a VM.

VNET Links: Each edge $e = (i, j) \in E$ is a prospective link between VNET daemons. e has a real-valued capacity c_e which is the bandwidth that the edge can carry in that direction. This is the available bandwidth between two Hosts (the ones running VNET daemons i and j) reported by Wren.

VNET Paths: A path, $p(i, j)$, between two VNET daemons $i, j \in H$ is defined as an ordered collection of links in E, $\langle(i, v_1), (v_1, v_2), ..., (v_n, j)\rangle$, which are the set of VNET links traversed to get from VNET daemon i to j given the current forwarding rules and topology, $v_1, \ldots v_n \in H$. P is the set of all paths.

VM Mapping: V is the set of VMs in the system, while M is a function mapping VMs to daemons. $M(k) = l$ if VM $k \in V$ is mapped to Host $l \in H$.

VM Connectivity: We are also given a set of ordered 3-tuples $A = (S, D, C)$. Any tuple, $A(s_i, d_i, c_i)$, corresponds to an entry in the traffic load matrix supplied by VTTIF. More specifically, consider two VMs, $k, m \in V$, where $M(k) = s_i$ and $M(m) = d_i$, then c_i is the traffic matrix entry for the flow from VM k to VM m.

Configurations: A configuration $CONF = (M, P)$ consists of the VM to VNET daemon mapping function M and the set of paths P among the VNET daemons needed to assure the connectivity of the VMs. The topology and forwarding rules for the VNET daemons follow from the set of paths.

Residual Capacity of a VNET Link: Each tuple, A_i, can be mapped to one of multiple paths, $p(s_i, d_i)$. Once a configuration has been determined, each VNET link $e \in E$ has a real-valued residual capacity rc_e which is the bandwidth remaining unused on that edge.

Bottleneck Bandwidth of a VNET Path: For each mapped paths $p(s_i, d_i)$ we define its bottleneck bandwidth, $b(p(s_i, d_i))$, as $(min(cr_e)).\forall e \in p(s_i, d_i)$.

Optimization Problem: We want to choose a configuration $CONF$ which maps every VM in V to a VNET daemon, and every input tuple A_i to a network path $p(s_i, d_i)$ such that the total bottleneck capacity on the VNET graph,

$$\sum_{p \in P} b(p(s_i, d_i)) \quad (1)$$

is maximized or minimized subject to the constraint that

$$\forall e \in E : rc_e \geq 0 \quad (2)$$

The intuition behind maximizing the residual bottleneck capacity is to leave the most room for the application to increase performance within the current configuration. Conversely, the intuition for minimizing the residual bottleneck capacity is to increase room for other applications to enter the system.

This problem is NP-complete by reduction from the edge disjoint path problem [3]. [2]

4.2 A greedy heuristic solution

In an online system of any scale, we are unlikely to be able to enumerate all possible configuration to choose a good one. Our approach is necessarily heuristic and is based on a greedy strategy with two sequential steps: (1) find a mapping from VMs to Hosts, and (2) determine paths for each pair of communicating VMs.

4.2.1 A greedy heuristic mapping VMs to Hosts

VADAPT uses a greedy heuristic algorithm to map virtual machines onto physical hosts. The input to the algorithm is the application communication behavior as captured by VTTIF and available bandwidth between each pair of VNET daemons, as reported by Wren, both expressed as adjacency lists.

The algorithm is as follows:

1. Generate a new VM adjacency list which represents the traffic intensity between VNET daemons that is implied by the VTTIF list and the current mapping of VMs to hosts.

2. Order the VM adjacency list by decreasing traffic intensity.

3. Extract an ordered list of VMs from the above with a breadth first approach, eliminating duplicates.

4. For each pair of VNET daemons, find the maximum bottleneck bandwidth (the widest path) using the adapted Dijkstra's algorithm described in Section 4.2.3.

5. Order the VNET daemon adjacency list by decreasing bottleneck bandwidth.

6. Extract an ordered list of VNET daemons from the above with a breadth first approach, eliminating duplicates.

7. Map the VMs to VNET daemons in order using the ordered list of VMs and VNET daemons obtained above.

8. Compute the differences between the current mapping and the new mapping and issue migration instructions to achieve the new mapping.

4.2.2 A greedy heuristic mapping communicating VMs to paths

Once the VM to Host mapping has been determined, VADAPT uses a greedy heuristic algorithm to determine a path for each pair of communicating VMs. The VNET links and forwarding rules derive from the paths. As above VADAPT uses VTTIF and Wren outputs expressed as adjacency lists as inputs.

The algorithm is as follows:

1. Order the set A of VM to VM communication demands in descending order of communication intensity (VTTIF traffic matrix entry).

2. Consider each 3-tuple in the ordered set A, making a greedy mapping of it onto a path. The mapping is on the current residual capacity graph G and uses an adapted version of Dijkstra's algorithm described in Section 4.2.3. No backtracking is done at this stage.

4.2.3 Adapted Dijkstra's algorithm

We use a modified version of Dijkstra's algorithm to select a path for each 3-tuple that has the maximum bottleneck bandwidth. This is the "select widest" approach. Notice that as there is no backtracking, it is quite possible to reach a point where it is impossible to map a 3-tuple at all. Furthermore, even if all 3-tuples can be mapped, the configuration may not minimize/maximize Equation 1 as the greedy mapping for each 3-tuple doesn't guarantee a global optimal.

Dijkstra's algorithm solves the single-source shortest paths problem on a weighted, directed graph $G = (H, E)$. We have created a modified Dijkstra's algorithm that solves the single-source widest paths problem on a weighted directed graph $G = (H, E)$ with a weight function $c : E \rightarrow \mathbb{R}$ which is the available bandwidth in our case.

As in Dijkstra's algorithm we maintain a set U of vertices whose final widest-path weights from the source u have already been determined. That is, for all vertices $v \in U$, we have $b[v] = \gamma(u, v)$, where $\gamma(u, v)$ is the widest path value from source u to vertex v. The algorithm repeatedly selects the vertex $w \in H - U$ with the largest widest-path estimate, inserts w into U and relaxes (we slightly modify the original Relax algorithm) all edges leaving w. Just as in the

[2] The proof is available on virtuoso.cs.northwestern.edu

implementation of Dijkstra's algorithm, we maintain a priority queue Q that contains all the vertices in $H - U$, keyed by their b values.

Similar to Dijkstra's algorithm we initialize the widest path estimates and the predecessors by the following procedure.

```
**Procedure** $Initialize(G, u)$
for each $vertex\ v\ \epsilon\ H[G]$ do
    {
    $b[v] \leftarrow 0$
    $\pi[v] \leftarrow NIL$
    }
end for
$b[u] \leftarrow \infty$
```

The modified process of relaxing an edge (w, v) consists of testing whether the bottleneck bandwidth decreases for a path from source u to vertex v by going through w, if it does, then we update $b[v]$ and $\pi[v]$.

```
**Procedure** $ModifiedRelax(w, v, c)$
if $b[v] < min(b[w], c(w, v))$ then
    {
    $b[v] \leftarrow min(b[w], c(w, v))$
    $\pi[v] \leftarrow w$
    }
end if
```

We can very easily see the correctness of ModifiedRelax. After relaxing an edge (w, v), we have $b[v] \geq min(b[w], c(w, v))$. As, if $b[v] < min(b[w], c(w, v))$, then we would set $b[v]$ to $min(b[w], c(w, v))$ and hence the invariant holds. Further, if $b[v] \geq min(b[w], c(w, v))$ to begin with, then we do nothing and the invariant still holds.

The following is the adapted version of Dijkstra's algorithm to find the widest path for a single tuple.

```
**Procedure** $AdaptedDijkstra(G, c, u)$
$Initialize(G, u)$
$U \leftarrow \emptyset$
$Q \leftarrow H[G]$
while $Q \neq \emptyset$ do {loop invariant: $\forall\ v\ \epsilon\ U,\ b(v) = \gamma(u, v)$}
    {
    $w \leftarrow ExtractMax(Q)$
    $U \leftarrow U \cup w$
    for each $vertex\ v\ \epsilon\ Adj[w]$ do
        {
            $ModifiedRelax(w, v, c)$
        }
    end for
    }
end while
```

4.2.4 Correctness of adapted Dijkstra's algorithm

Similar to the proof of correctness for Dijkstra's shortest paths algorithm, we can prove that the adapted Dijkstra's algorithm is correct by proving by induction on the size of set U that the invariant, $\forall v \in U$, $b[v] = \gamma(u, v)$, always holds.

Base case: Initially $U = \emptyset$ and the invariant is trivially true.

Inductive step: We assume the invariant to be true for $|U| = i$.

Proof: Assuming the truth of the invariant for $|U| = i$, we need to show that it holds for $|U| = i + 1$ as well.

Let v be the $(i+1)^{th}$ vertex extracted from Q and placed in U and let p be the path from u to v with weight $b[v]$. Let w be the vertex just before v in p. Since only those paths to vertices in Q are considered that use vertices from U, $w \in U$ and hence by the inductive step we have $b[w] = \gamma(u, w)$.

Next, we can prove that p is the widest path from u to v by contradiction. Let us assume that p is not the widest path and instead p^* is the widest path from u to v. Since this path connects a vertex in U to a vertex in $H - U$, there must be a first edge, $(x, y) \in p^*$ where $x \in U$ and $y \in H - U$. Hence the path p^* can now be represented as $p_1.(x, y).p_2$. By the inductive hypothesis $b[x] = \gamma(u, x)$ and since p^* is the widest path, it follows that $p_1.(x, y)$ must be the widest path from w to y, as if there had been a path with higher bottleneck bandwidth, that would have contradicted the optimality of p^*. When the edge x was placed in U, the edge (x, y) was relaxed and hence $b[y] = \gamma(u, y)$. Since v was the $(i + 1)^{th}$ vertex chosen from Q while y was still in Q, it implies that $b[v] \geq b[y]$. Since we do not have any negative edge weights and $\gamma(s, v)$ is the bottleneck bandwidth on p^*, that combined with the previous expression gives us bottleneck bandwidth of $p^* \leq b[v]$ which is the bottleneck bandwidth of path p. This contradicts our first assumption that path p^* is wider than path p.

Since we have proved that the invariant holds for the base case and that the truth of the invariant for $|U| = i$ implies the truth of the invariant for $|U| = i+1$, we have proved the correctness of the adapted Dijkstra's algorithm using mathematical induction.

4.3 A simulated annealing heuristic solution

Simulated annealing [5] (SA) is a probabilistic evolutionary method that is well suited to solving global optimization problems, especially if a good heuristic is not known. SA's ability to locate a good, although perhaps nonoptimal solution for a given objective function in the face of a large search space is well suited to our problem. Since the physical layer and VNET layer graphs in our system are fully connected there are a great many possible forwarding paths and mappings. Additionally, as SA incrementally improves its solution with time, there is some solution available at all times.

The basic approach is to start with some initial solution to the problem computed using some simple heuristic such as the adapted Dijkstra based heuristic described above. SA iterations then attempt to find better solutions by perturbing the current solution and evaluating its quality using a cost function. At any iteration, the system state is the set of prospective solutions. The random perturbations of the SA algorithm make it possible to explore a diverse range of the search space including points that may appear sub-optimal or even worse then previous options but may lead to better solutions later on. The probability of choosing options that are worse than those in the present iteration is reduced as the iterations proceed, focusing increasingly on finding better solutions close to those in the current iteration. In physical annealing, this probability results from the present temperature of a metal that is slowly cooled.

4.3.1 Perturbation function

The role of the perturbation function (PF) is to find neighbors of the current state that are then chosen according to a probability function $P(dE, T)$ of the energy difference $dE = E(s') - E(s)$ between the two states, and of a global time-varying parameter T (the temperature). The probability function we use is $e^{dE/T}$ if dE is negative, 1 otherwise. As iterations proceed T is decreased which reduces the probability of jumping into states that are worse than the current state.

Given a configuration $CONF = (M, P)$, where P is a set of forwarding paths $p(i, j)$ and each $p(i, j)$ is a sequence of $k_{i,j}$ vertices $v_i, v_1, v_2, ..., v_j$, the perturbation function selects a neighbor $N(CONF)$ of the current configuration with the following probabilities: For each $p(i, j) \in P$:

1. With probability $1/3$ PF adds a random vertex v_r into the path sequence where $v_r \in V$ and $\notin p(i, j)$. Note that the set V consists of all potential physical nodes which are running VNET and hence are capable of routing any VNET traffic. This step attempts to modify each path by randomly adding a potential overlay node in the existing forwarding path.

2. With probability $1/3$ PF deletes a random vertex v_r from the path sequence where $v_r \in p(i, j)$.

3. With probability $1/3$ PF swaps two nodes v_x and v_y where $x \neq y$ and $v_x, v_y \in p(i, j)$.

4.3.2 Cost evaluation function

The cost evaluation function CEF computes the cost of a configuration C using Equation 1. After a neighbor $N(C)$ is found using the perturbation function, a cost difference $CEF(N(C)) - CEF(C)$ is computed. This is the energy difference used to compute the future path in the simulated annealing approach using a probability $e^{(CEF(N(C)) - CEF(C))/t}$ if the difference is negative, 1 otherwise. As iterations proceed and temperature decreases, the SA algorithm finally converges to the best state it encounters in its search space.

4.3.3 Algorithm

The following is our simulated annealing algorithm:

```
Procedure $anneal(C_0, t_0, a, PF, CEF)$
t = t_0
$C = C_0$
while $t > t_{min}$ do
    for $i = 0$ to $iterationCount$ do
        if $staleIterations = staleThreshold$ then
            Perturb Mapping
            $stateIterations = 0$
        end if
        $N = PF(C)$
        $dE = CEF(N) - CEF(C)$
        if $dE < 0$ then
            With probability $e^{dE/t}$ let $C = N$
        else
            let $C = N$
            $staleIterations = 0$
        end if
        $staleIterations = staleIterations + 1$
    end for
    $t = t.a$
end while
```

The important factors which affect the performance of the simulated annealing algorithm are (1) the temperature schedule, which decides how temperature is modified after each set of iterations, and (2) the perturbation function PF. The choice of these perturbation is especially critical because it must be able to find neighbors with similar energy levels as previous states. Our evaluation has shown empirically that the PF described above is very effective.

Perturbing the mapping: On a typical iteration, our algorithm only perturbs the current forwarding paths. To also explore new mappings of the VMs to different VNET hosts, we also perturb that mapping. However, as perturbing a mapping effectively resets the forwarding paths, we perturb the mappings with a lower frequency. As the temperature decreases, the change in CEF for new configurations also falls. Our SA algorithm keeps track of CEF and after a certain $stalenessfactor$ when the CEF does not change for a certain number of iterations, we perturb the mapping. A non-changing CEF suggests that the SA algorithm has converged to a good solution for the particular mapping. After each iteration, we store its best solution. We can output a solution at any time.

Multi-constrained Optimization: One of the strengths of the simulated annealing technique is that it is straightforward to include complex objective functions. To do so, only the objective function evaluation needs to be changed to return the appropriate cost for an objective function and the annealing process automatically explores the search space appropriately using this new cost function.

4.4 Performance

Because we have not yet coupled the entire real-time toolchain, our evaluation is done in simulation, using Wren measurements collected from observing VNET data to the extent possible. We also evaluate our algorithms by posing a challenging adaptation problem, and evaluate their scalability using a large-scale problem. In each scenario the goal is to generate a configuration consisting of VM to Host mappings and paths between the communicating VMs that maximizes the total residual bottleneck bandwidth (Section 4.1). We compare the greedy heuristic (GH), simulated annealing approach (SA) and simulated annealing with the greedy heuristic solution as the starting point (SA+GH). In addition at all points in time we also maintain the best solution found so far with (SA+GH), we call this (SA+GH+B), where ’B’ indicates the best solution so far. The W&M and NWU setup had a solution space small enough to enumerate all possible configurations to find the optimal solution.

4.4.1 Wren measurements for William and Mary (W&M) and Northwestern (NWU)

We have created a testbed of Wren-enabled machines : two at William and Mary and two at Northwestern as shown in Figure 6. We have successfully run VNET on top of Wren on these systems with Wren using VM traffic to characterize the network connectivity, as shown in Figure 4. At the same time Wren provides its available bandwidth matrix, VTTIF provided the (correct) application topology matrix. The full Wren matrix is used in Section 4.4.2.

4.4.2 Adaptation in W&M and NWU testbed

We evaluated our adaptation algorithms for an application running inside of VMs hosted on the W&M and NWU testbed in simulation. The VMs were running the NAS MultiGrid benchmark. The lower part of Figure 6 shows the application topology inferred by VTTIF for a 4 VM NAS MultiGrid benchmark. The thickness of the arrows are directly proportional to the bandwidth demand in that direction.

Figure 7 shows the performance of our algorithms as a function of time. The two flat lines indicate the heuristic (GH) performance and the optimal cost of the objective function (evaluated by hand). Since the solution space is small with 12 possibilities for the VM to VNET mapping, we were able to enumerate all possible configurations and thus determine the optimal solution. The optimal mapping as found by the annealing algorithm is $VM1 \rightarrow 2$, $VM2 \rightarrow 4$, $VM3 \rightarrow 3$, $VM4 \rightarrow 1$ with an optimal CEF value of 605.66.

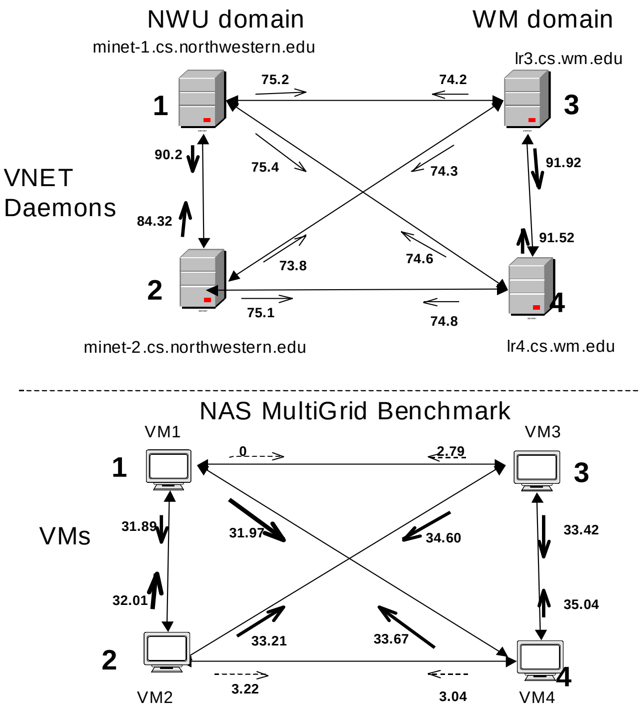


Figure 6. The Northwestern / William and Mary testbed, All numbers are in Mb/sec.

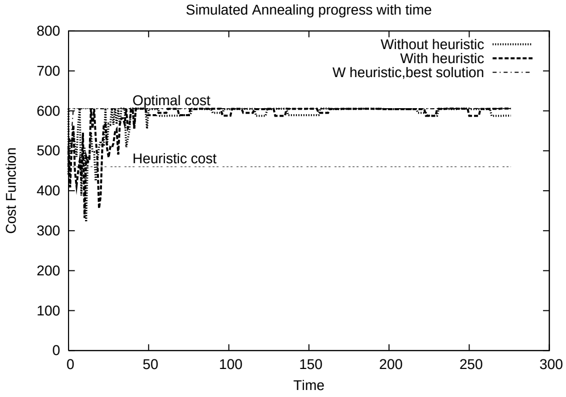


Figure 7. Adaptation performance while mapping 4 VM all-to-all application onto NWU / W&M testbed.

There is a curve for the simulated annealing algorithm,

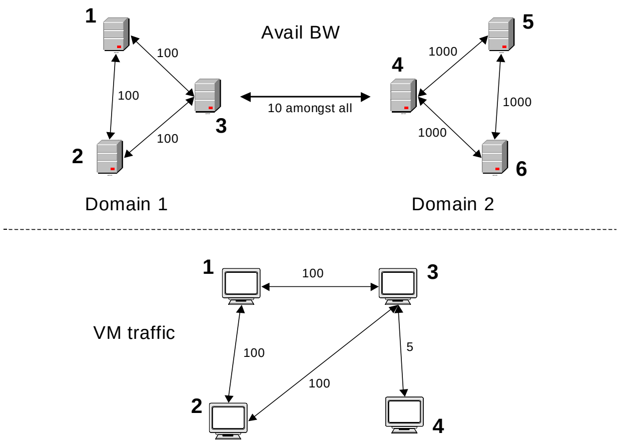
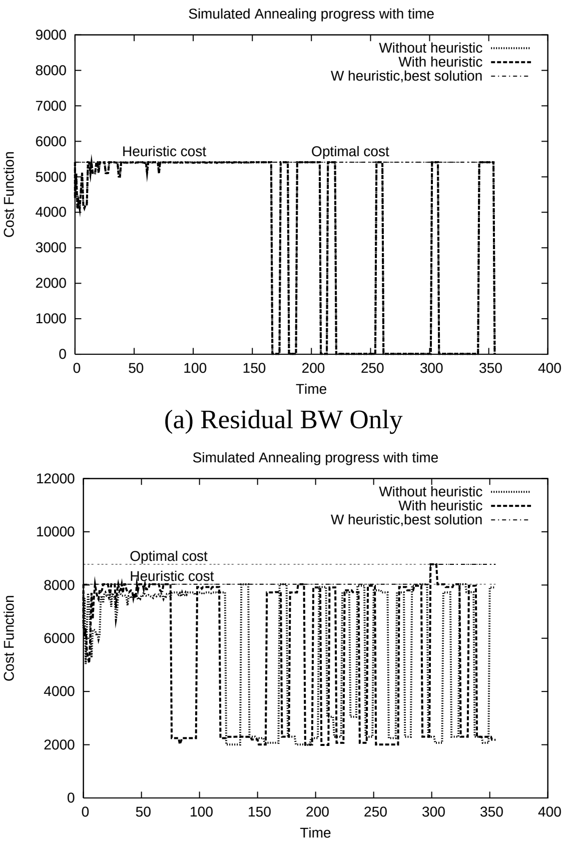


Figure 8. A Challenging scenario thet requires a specific VM to Host mapping for good performance.

SA+GH (annealing algorithm starting with heuristic as the initial point) and the best cost reached so far, showing their values over time. We see that the convergence rate of SA is crucial to obtaining a good solution quickly. Notice that SA is able to find close to optimal solutions in a reasonably short time, while GH completes almost instantaneously, but is not able to find a good solution. SA+GH performs slightly better than SA. Note that the graph shows, for each iteration, the best value of the objective function of that iteration. SA+GH+B shows the best solution of all the iterations up to the present one by SA+GH.

4.4.3 Challenge

We also designed a challenging scenario, illustrated in Figure 8, to test our adaptation algorithms. The VNET node topology consists of two clusters of three machines each. The domain 1 cluster has 100 Mbps links interconnecting the machines, while domain 2 cluster has 1000 Mbps links. The available bandwidth on the link connecting the two domains is 10 Mbps. This scenario is similar to a setup consisting of two tightly coupled clusters connected to each other via WAN. The lower part of the figure shows the VM configuration. VMs 1, 2 and 3 communicate with a much higher bandwidth as compared to VM 4. An optimal solution for this would be to place VMs 1,2 and 3 on the three VNET nodes in domain 2 and place VM 4 on a VNET node in domain 1. The final mapping reported by GH is $VM1 \rightarrow 5$, $VM2 \rightarrow 4$, $VM3 \rightarrow 6$, $VM4 \rightarrow 1$. The final mapping reported by SA+GH is $VM1 \rightarrow 4$, $VM2 \rightarrow 5$, $VM3 \rightarrow 6$, $VM4 \rightarrow 1$. Both are optimal for the metric described before with a final CEF value of 5410.



(a) Residual BW Only

(b) Residual BW and Latency

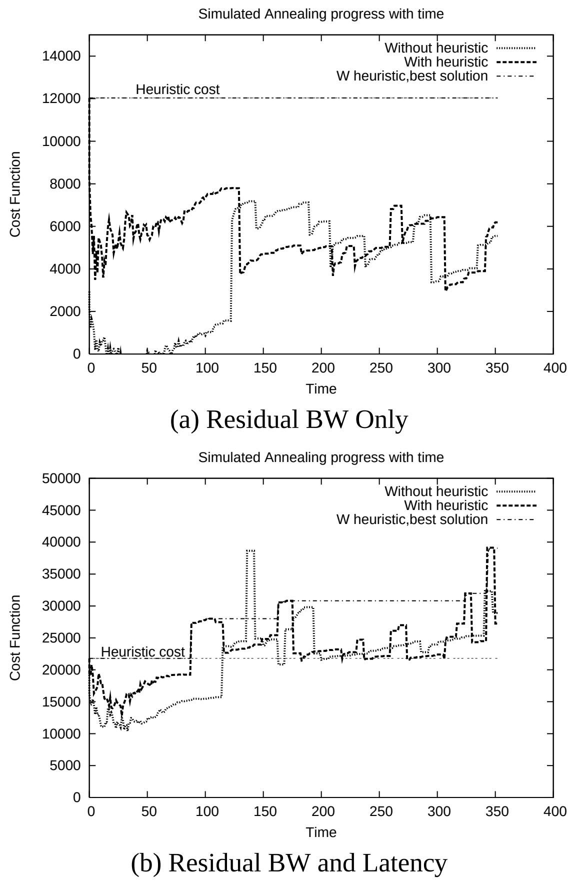
Figure 9. Adaptation performance while mapping 6 VM all-to-all application onto the challenging scenario.

For this scenario, both, GH and SA are able to find the optimal mappings quickly. Figure 9(a) illustrates the performance of our adaptation algorithms. The physical and application topologies have been constructed so that only one valid solution exists. We see that GH, SA, and SA+GH, all find the optimal solution quickly with very small difference in their performance. The large fluctuations in the objective function value for SA curves is due to the occasional perturbation of VM to Host mapping. If a mapping is highly sub-optimal, the objective function value drops sharply and remain such until a better mapping is chosen again.

Multi-constraint Optimization: In this scenario, we also use the annealing algorithm to perform multi-constrained optimization described previously In Figure 9(b), we show the performance of our algorithms with an objective function that takes into account, both, bandwidth and latency. Specifically, we have changed Equation 1 to be

$$\sum_{p \in P} b(p(s_i, d_i)) + \frac{c}{l(p(s_i, d_i))} \tag{3}$$

where $l(p)$ is the path latency for path p and c is a constant. This penalizes the paths with large latencies. We see that SA and SA+GH find better solutions than GH. GH provides a



(a) Residual BW Only

(b) Residual BW and Latency

Figure 10. Adaptation performance while mapping 8 VM all to all to 32 VNET daemons running on 256 node BRITE topology.

good starting point for SA which further explores the search space to improve the solution based on the defined multi-constraint objective.

4.4.4 Large topology

To study scalability of our adaptation algorithms we generated a 256 node BRITE [9] physical topology. The BRITE topology was generated using the Waxman Flat-router model with a uniformly varying bandwidth from 10 to 1024 units. Each node has an out-degree of 2. In this topology, we chose 32 hosts at random to run VNET daemons, hence each is a potential VM host.

A VNET link is a path in the underlying BRITE physical topology. We calculated the bandwidths for the VNET overlay links as the bottleneck bandwidths of the paths in the underlying BRITE topology connecting the end points of the VNET link.

Figure 10 shows the performance of our algorithms adapting a 8 VM patterns application communicating with a ring topology to the available network resources. It illustrates the point that the simple greedy heuristic is more suited to smaller scenarios, while simulated annealing is best used when the quality of the solution is most important and computation time can be traded for the same.

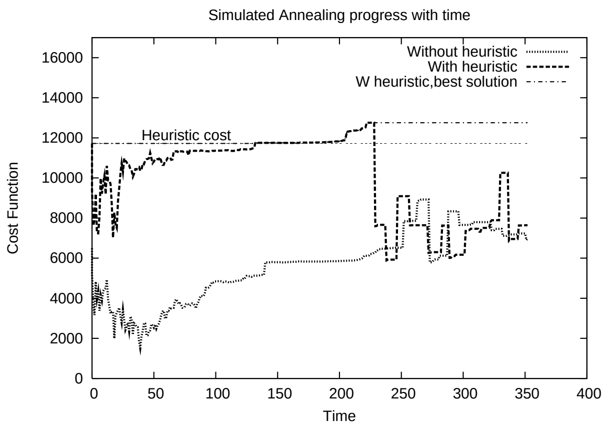


Figure 11. Adaptation performance while mapping 8 VMs with ring topology to 32 VNET daemons.

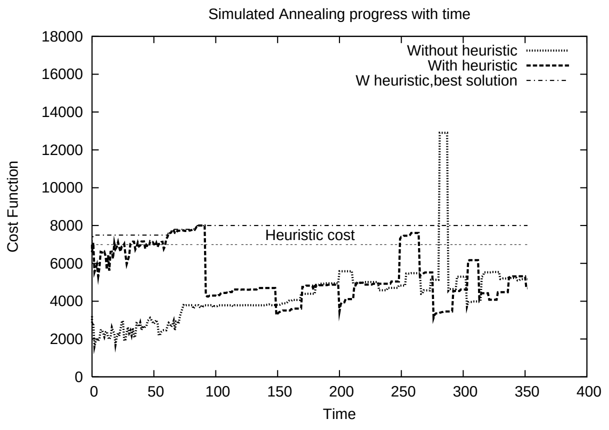


Figure 12. Adaptation while mapping 4 VMs running the NAS MultiGrid benchmark to 32 VNET daemons

GH completes very quickly and produces a solution, which, at this point in time, cannot be compared to the optimal as we do not know the optimal solution. Simulated annealing on the other hand takes a much longer time, but produces a much better result at the end of it. Figure 10 shows the scenario until the point where its solution is not as good as GH, however, given more time and compute resources, it would complete much faster producing a much closer to optimum solution. However, the same cannot be noted from the figure.

Figure 10(b) shows the performance using the objective function of Equation 3. Here, the situation is reversed, not surprisingly given that GH does not consider latency at all. These results indicate that simulated annealing is very effective in finding good solutions for larger scale problems for complex objective functions when it may be difficult to devise appropriate heuristics.

Figure 11 shows the performance when running a 8 VM patterns application with ring communication topology mapped onto 32 VNET daemons. In this case we note that the SA+GH is able to do better than plain SA and GH. So that re-enforces the point that a reasonable starting heuristic goes a long way in improving the performance of SA. This effect can also be see in Figure 12 in which 4 VMs running the NAS MultiGrid benchmark are mapped onto 32 VNET daemons.

5 Conclusions

We have described how the Virtuoso and Wren systems may be integrated to provide a virtual execution environment that simplifies application portability while providing the application and resource measurements required for transparent optimization of application performance. We have described extensions to the Wren passive monitoring system that support online available bandwidth measurement and export the results of those measurements via a SOAP interface. Our results indicate that this system has low overhead and produces available bandwidth observations while monitoring bursty VNET traffic. VADAPT, the adaptation component of Virtuoso uses this information provided by Wren along with application characteristics provided by VTTIF to dynamically configure the application, maximizing its performance. We formalize the adaptation problem, and compare two heuristic algorithms as solutions to this NP-hard problem. We found the greedy heuristic to perform as well or better than the simulated annealing approach, however, if the heuristic was taken as the starting point for simulated annealing it performed much better than the greedy heuristic.

References

[1] Figueiredo, R., Dinda, P. A., and Fortes, J. A case for grid computing on virtual machines. In *Proceedings of the 23rd International Conference on Distributed Computing Systems (ICDCS 2003)* (May 2003).

[2] Gupta, A., and Dinda, P. A. Inferring the topology and traffic load of parallel programs running in a virtual machine environment. In *Proceedings of the 10th Workshop on Job Scheduling Strategies for Parallel Processing (JSPPS 2004* (June 2004). To Appear.

[3] Karp, R. *Compexity of Computer Computations*. Miller, R.E. and Thatcher, J.W. (Eds.). Plenum Press, New York, 1972, ch. Reducibility among combinatorial problems, pp. 85–103.

[4] Keahey, K., Doering, K., and Foster, I. From sandbox to playground: Dynamic virtual environments in the grid. In *Proceedings of the 5th International Workshop on Grid Computing* (November 2004).

[5] Kirkpatrick, S., Gelatt, C. D., and Vecchi, M. P. Optimization by simulated annealing. *Science, Number 4598, 13 May 1983 220, 4598* (1983), 671–680.

[6] Lange, J. R., Sundararaj, A. I., and Dinda, P. A. Automatic dynamic run-time optical network reservations. In *Proceedings of the Fourteenth International Symposium on High Performance Distributed Computing (HPDC)* (July 2005). To appear.

[7] Lin, B., and Dinda, P. Vsched: Mixing batch and interactive virtual machines using periodic real-time scheduling. Tech. Rep. NWU-CS-05-06, Department of Computer Science, Northwestern University, April 2005.

[8] Man, C. L. T., Hasegawa, G., and Murata, M. A merged inline measurement method for capacity and available bandwidth. In *Passive and Active Measurement Workshop (PAM2005)* (2005), pp. 341–344.

[9] Medina, A., Lakhina, A., Matta, I., and Byers, J. Brite: An approach to universal topology generation. In *Proccedings of International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunications Systems (MASCOTS)* (August 2001).

[10] Prasad, R., Murray, M., Dovrolis, C., and Claffy, K. Bandwidth estimation: Metrics, measurement techniques, and tools. In *IEEE Network* (June 2003).

[11] Ribeiro, V., Riedi, R. H., Baraniuk, R. G., Navratil, J., and Cottrell, L. pathChirp:Efficient Available Bandwidth Estimation for Network Paths. In *Passive and Active Measurement Workshop (PAM2003)* (2003).

[12] Shakkottai, S., Brownlee, N., and kc claffy. A study of burstiness in tcp flows. In *Passive and Active Measurement Workshop (PAM2005)* (2005), pp. 13–26.

[13] Shoykhet, A., Lange, J., and Dinda, P. Virtuoso: A system for virtual machine marketplaces. Tech. Rep. NWU-CS-04-39, Department of Computer Science, Northwestern University, July 2004.

[14] Sundararaj, A., and Dinda, P. Towards virtual networks for virtual machine grid computing. In *Proceedings of the 3rd USENIX Virtual Machine Research And Technology Symposium (VM 2004)* (May 2004). To Appear. Earlier version available as Technical Report NWU-CS-03-27, Department of Computer Science, Northwestern University.

[15] Sundararaj, A., Gupta, A., , and Dinda, P. Increasing application performance in virtual environments through run-time inference and adaptation. In *Proceedings of the 14th IEEE International Symposium on High Performance Distributed Computing (HPDC)* (July 2005). To Appear.

[16] Sundararaj, A., Gupta, A., and Dinda, P. Dynamic topology adaptation of virtual networks of virtual machines. In *Proceedings of the Seventh Workshop on Langauges, Compilers and Run-time Support for Scalable Systems (LCR)* (October 2004).

[17] Zangrilli, M., and Lowekamp, B. B. Using passive traces of application traffic in a network monitoring system. In *Proceedings of the Thirteenth IEEE International Symposium on High Performance Distributed Computing (HPDC 13)* (June 2004), IEEE.

[18] Zangrilli, M., and Lowekamp, B. B. Applying principles of active available bandwidth algorithms to passive tcp traces. In *Passive and Active Measurement Workshop (PAM 2005)* (March 2005), LNCS, pp. 333–336.