Scaling Network Emulation using Topology Partitioning

Ken Yocum, Ethan Eade, Julius Degesys
David Becker, Jeff Chase and Amin Vahdat *
TR CS-2003-01
Department of Computer Science
Duke University
{grant,eade,degesys,becker,chase,vahdat}@cs.duke.edu

Abstract

Network systems and services are growing in scale and complexity; evaluation infrastructures must scale to study these systems under realistic Internet-like scenarios. This paper studies topology partitioning*, assigning disjoint pieces of the network topology across processors, as a technique to increase emulation capacity with increasing hardware resources. We develop* static *methods to create partitions based on expected communication across the topology. Because the quality of the partition depends on the network topology and communication patterns, we also create a dynamic scheme which repartitions the topology across processors as communication patterns evolve. Our evaluation methodology quantifies the* efficiency *of the emulation to capture the communication overhead of the distributed computation. We implement both strategies in ModelNet, a large-scale network emulator, across different topologies and communication patterns. Results show that current graph partitioning algorithms can double the efficiency of the emulation for Internet-like topologies relative to random partitioning. These techniques allow researchers to study distributed systems across large, realistic network topologies, and apply equally to scaling network simulators.*

Keywords: *Network emulation, simulation, parallel and distributed systems, graph partitioning.*

1 Introduction

Today, techniques for building highly scalable and available network services and systems are receiving increasing attention from both the research and industrial communities. The complexity of these network systems requires accurate evaluation techniques. An network emulator subjects the communication of unmodified applications running on unmodified operating systems to the characteristics of a target large-scale topology. Recent work in network emulation [14, 15, 12] has shown that it is an attractive alternative to simulation for running controlled large-scale experiments. In contrast to simulation, emulation is performed in real time, with the goal of making the service's operation across the emulated network indistinguishable from running across the Internet.

Scalability is the primary bottleneck to the utility of emulation; the emulator must scale with the network system to be evaluated. Earlier work demonstrates that a single modern PC can accurately emulate 50-100 thousand packets per second, depending on the characteristics of the target network topology. In this paper, we apply techniques to scale network emulation capacity with additional machines. Many of the traditional distributed computing issues apply to scaling network emulation. In particular we develop techniques to partition emulation work across a set of nodes. In this paper, we treat network emulation as a large-scale distributed computation of packet wait times.

The job of a network emulator (or simulator) is to correctly delay the incoming network traffic according the characterstics of the network topology. The total number of packets which are correctly delayed per unit time is the emulation capacity of the system. The unit of work in network emulation is receiving a packet and delaying (or dropping) it appropriately for each hop in the path between the source and destination in the emulated network. We study techniques that *partition* the topology across the machines, called *cores*, responsible for performing emulation. This centralizes the state associated with a particular emulated network link (e.g., dynamically changing queue occupancy) at a single core, ensuring accuracy for all packets traversing that link. However, packets that must make their way between partitions require *cross-core* communication to forward state from one machine to another. Cross-core communication is the principal source of overhead when using multiple cores. Effectively, it is a tax on every emulated packet that wastes CPU and network resources. The quality of a partition depends on the network topology, the num-

*This work is supported in part by the U.S. National Science Foundation (CCR-00-82912, EIA-9972879, and EIA-9870728), Network Appliance, Cisco Systems and IBM..

ber of cores, and the communication pattern of the network system. A good partitioning minimizes cross-core communication.

This paper makes the following contributions. First, we develop algorithms that allow us to partition the graph based on static *expected* communication patterns. Second, we reduce the stress on the physical network links of our emulator through the use of *payload caching*. We develop an analytic model of payload caching and verify its utility empirically. Third, we compare two graph partitioning algorithms to random partitions with respect to their ability to minimize cross-core communication and balance load. Fourth, we instrument a large-scale network emulator to collect live traffic flows and dynamically repartition the target topology "on the fly."

All techniques are integrated into ModelNet [14], a large-scale network emulator. We have completed analytical and empirical evaluations of these techniques. We evaluate our static partitioning scheme across a range of toy and Internet-like topologies while varying traffic patterns. We believe that our general approach is applicable to scaling network emulation and simulation in the general case.

2 Related Work

The focus of this paper is scaling network emulation by intelligently distributing the work across compute nodes. This is in contrast to techniques that simplify the studied network system to reduce computational costs. For example, one can modify the target topology to reduce the cost of emulation. Distillation in ModelNet reduces average path length of the target topology through graph transforms [14]. The SHRiNK work simulates large networks with smaller scale replicas [8]. Selective abstraction was used to reduce the computational cost of simulating multicast protocols [5]. These are important complementary techniques to the topology partitioning and replication studied here.

Using multiprocessors or a machine cluster has been explored in network simulation efforts. Work on parallel distributed Ns (PDNS) adapted Ns to run over multiple workstations, but performed partitioning by hand [11]. Some network simulators and emulators are implemented over parallel discrete event simulation (PDES) engines that run on tightly-coupled multiprocessors [2, 10, 12]. The GloMoSim wireless network simulator explored simple static partitioning of a wireless network environment [18]. The Genesis network simulator partitions the network across processors; each partition contains a model to simulate the other partitions. The model is tuned through successive iterations of the same simulation time interval until each partition produces accurate results; partitions are created by hand [13].

3 Network Emulation

A network emulation is defined by the application generating network traffic and a *target* network topology. The goal of scaling network emulation is to increase emulation capacity, accurately emulated application traffic. In an emulation, applications generate network traffic and the hardware emulates a target network topology. The general problem is specified by: a target topology, a communication pattern, and physical hardware resources (a cluster of workstations or multiprocessor). We call each physical processor a *core* node.

The target topology is a graph G_t that is weighted and undirected, $G(V, E)$. Each edge e represents a link in the target topology, and has a specified bandwidth, latency, loss probability, and queue management policy (drop tail, RED, etc.,). In this paper we refer to edges as *pipes*, as they also represent packet queues in the network emulator. Costs in this model are based on the flow of packets into the emulator from the application. There are three principal costs: λ_{edge}, λ_{hops}, and λ_{xc}. Traffic entering the emulation from applications is defined as *edge* traffic, λ_{edge}. λ_{hop} is the total number of packet hops that core nodes emulate per second. *Cross-core* traffic, λ_{xc}, represents the communication cost of harnessing many processing elements to perform the emulation. The techniques in this paper try to maximize emulation capacity, λ_{edge}, for a given number of processors, target topology, and communication pattern. They map emulation work to processing elements at the granularity of edges in the target topology.

Partitioning divides the target topology among k core nodes. Each core is responsible for emulating that piece of the target topology. Figure 1 shows an example of partitioning as two 4 hop paths are traversed; pipe a is assigned to core A and pipe b to core B. In partitioning, all cross-core communication λ_{xc} consists of packets. Because emulations carry application traffic, they must store the bandwidth-delay product (BDP) of the target topology, and each partition is responsible for the BDP in that area of the network.

We use a simple metric, *effi ciency* (σ), to measure the quality of the topology partitioning for scaling network emulation. We want to maximize throughput from edge nodes; these packets represent the goodput of the emulation. At the same time we want to minimize the cross-core traffic; λ_{X_t} is the total number of packets core X processes. $\lambda_{X_{edge}}$ is the portion of λ_{X_t} that comes from edge nodes. $\lambda_{X_{xc}}$, the cross-core flow to core X, is $\lambda_{X_t} - \lambda_{X_{edge}}$. If k is the number of cores in the system, then $\sigma = \frac{\sum_{i=1}^{k} \lambda_{i_{edge}}}{\sum_{i=1}^{k} \lambda_{i_t}}$. This represents the percentage of packets dedicated to goodput. The goal of the techniques in this paper is to drive this measure toward one while maintaining an accurate emulation.

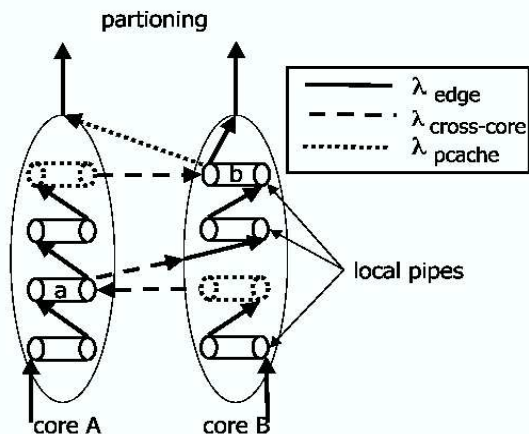


Figure 1. This figure shows a two core partitioned network emulation. The topology consists of two 4 pipe paths: one starts from core A, the other from core B. Pipes drawn with dotted lines are assigned to the other core, and labeled pipes are shared by the routes. The overhead of the network emulation consists of the number of packets that need to be sent between cores in order to be delivered to their destination, λ_{xc}.

We must deal with one other effect of cross-core traffic, increased physical link stress. A packet takes zero or more cross-core hops as it makes its way from source to destination in the target topology. Each cross-core hop consumes CPU at each core node, but it also consumes physical bandwidth between the core nodes in the cluster. For example, a single emulated 10Kb flow could physically consume 100Kb if each packet caused 10 cross-core hops. Payload caching is a technique that reduces the size of the cross-core packets.

4 Reducing Physical Link Stress via Payload Caching

Payload caching is a technique (loosely based on [17]) that reduces the per-byte overhead of cross-core communication when using partitioning. This section analytically and empirically explores the benefit of payload caching. Payload caching caches packet data on the "home" core; cross-core traffic consists only of packet headers. When the last hop on the path from source to destination is emulated, the packet header is sent to the home core to reunite with its data before being sent to the destination application. Payload caching reduces the bytes of cross-core traffic [1], but may increase the number of cross-core hops by one. In the worst case, payload caching could force every packet to take an extra cross-core hop, increasing the CPU load for each home core. In contrast, payload *shipping* transfers the entire packet between cores.

[1] With a 1500 byte MTU, payload caching reduces the number of bytes on the wire by an order of magnitude.

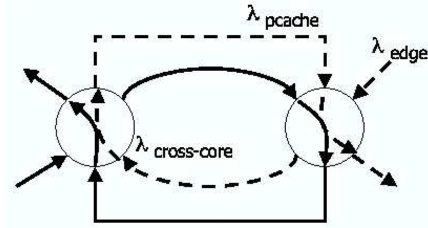
The following analytic model studies the costs and benefits of payload caching. The figure on the left in Figure 2 shows two cores and the flow of emulated traffic. Our model makes the following assumptions: the input flow, λ_{edge}, is uniform across cores, all cores forward 100% of their traffic, and each packet takes an identical number of cross-core hops. Each core forwards λ_{xc} to other cores, and, when payload caching is enabled, λ_{pcache} to other cores. In this model $\lambda_{edge} = \lambda_{xc} = \lambda pcache$. Note that this analysis is independent of the number of cores.

Payload caching trades increased CPU utilization for reduced link stress on the emulator's physical network. The right-hand graph in Figure 2 illustrates the case where both tactics are limited by the network bandwidth, parametrized to a gigabit link. Payload caching provides superior performance relative to packet shipping as the number of cross-core hops increases. In the CPU-limited case, the left-hand graph in Figure 3, payload caching is penalized for its extra hop and only until this cost is amortized across many cross-core hops (approximately 8) does payload caching approach packet shipping performance.

We validated the model by empirically measuring λ_{edge} using two ModelNet cores while increasing the cross-core traffic. To increase cross-core traffic, pipe ownership changes alternate along increasing path lengths. In our particular system, the bottleneck when using payload caching is the CPU, and the bottleneck when using payload shipping is the physical link. The right-hand graph in Figure 3 plots those two results from our model against our empirical measurements. The step-like plot from payload caching occurs because odd-numbered cross-core hops need an extra hop to return to the home core. Even-numbered cross-core hops are already home. Note that payload caching never penalizes performance relative to payload shipping. After 10 cross-core hops, payload caching yields a 20% performance gain, which increases with the number of cross-core hops. In general CPU performance climbs faster than network performance; we expect the benefit from payload caching to increase in the future.

5 Partitioning

This section formally defines graph partitioning and maps the general problem unto partitioning a target topology across a set of k physical core nodes. Graph partitioning is a difficult, long-standing computational problem. It has applications to VLSI design, sparse matrix-vector multiplication, and parallelizing scientific algorithms. With respect to emulation, a good partition on the target topology



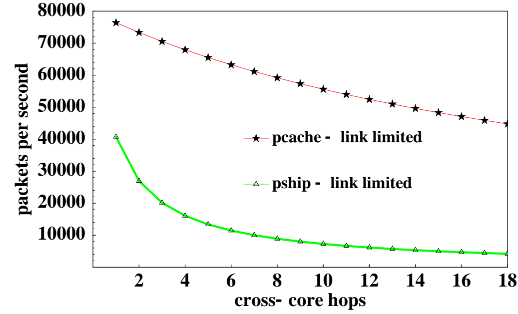
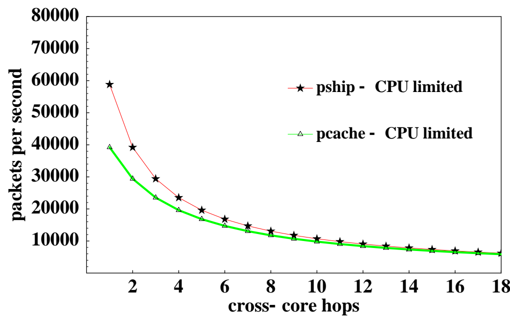


Figure 2. On the left, the analytic payload caching model shows cached packets, λ_{pcache}, returning home before exiting the emulator core. The graph on the right plots the model's predicted edge traffic, λ_{edge}, per core as the number of cross-core hops taken by each packet increases for payload caching (pcache) and payload shipping (pship). In this case both tactics are limited purely by link throughput (1 Gbps). With much smaller packets, payload caching significantly reduces link stress and achieves higher throughput than payload shipping.



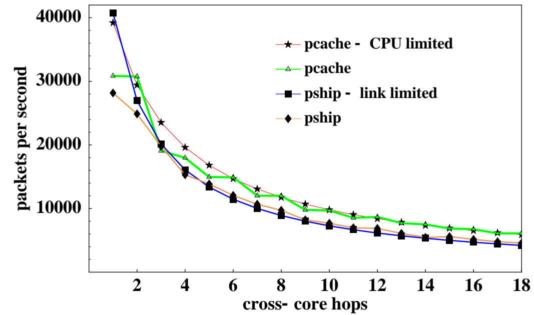


Figure 3. The left-hand graph illustrates the per-core edge traffic, λ_{edge}, as predicted by the analytic model when the limiting factor is the CPU (the maximum packets per second (PPS) is 117K per core). In this case the extra cross-core hop to return to the home core penalizes payload caching. The right graph plots bandwidth-limited payload shipping versus CPU-limited payload caching. Included on this plot are the results of experiments on a two core emulator. This shows that the model is both accurate and that payload caching only increases emulation capacity.

balances the load across the core nodes and minimizes the cross-core communication. Load balancing is important because an inaccurate emulation can result from an overloaded core node [14]. This is in contrast to mapping a scientific computation across a cluster, where a loaded processor may increase run time but not affect the answer's accuracy. Minimizing cross-core communication is important because it increases emulation efficiency and accuracy as cross-core hops may add real time delay to the emulated path.

The general k-way partitioning problem is described by a graph $G(V, E, W_V, W_E)$. Where W_V and W_E represent vertex and edge weights respectively. The output of partitioning G consists of subsets of *vertices*, $V_1, V_2, ..., V_k$, where $V_i \cup V_j = \emptyset$ for $j \neq i$, and $\bigcup_i V_i = V$. The goal is to balance the sum of vertex weights for each V_i, and minimize the sum of edge weights whose incident vertices belong to different partitions. This problem is NP-hard [4], however heuristics yield good partitions in practice [6].

Generic graph partitioning algorithms pose two issues when applied to network topology partitioning. First, graph partitioners provide a *vertex* partitioning. This is contrast to the *edge* partition resulting from the assignment of edges in the target topology, G_t, to core nodes. Second, this graph requires both vertex and edge weights. These weights represent core load and network flow respectively. The flow is unknown until the emulation runs. Section 5.3 describes a simple method to map G_t to a vertex-partitionable graph and weight the graph according to an *expected* communication pattern without running the emulation.

We first develop a simple model to explore the relation-

ship between increasing the number of cores and increasing the efficiency of the partition. We then describe two partitioning algorithms: k-cluster, which is based on the connectivity of the target topology, and METIS, a heuristic approach based on connectivity and network flow. This section ends with a description of a dynamic partitioning scheme which repartitions the topology "on the fly" during an emulation.

5.1 Analytic Partitioning Model

Emulation capacity can be scaled either by adding more core nodes or by increasing efficiency. The load on the cores depends on application traffic rates, λ_{edge}, application communication patterns that define the set of edges carrying traffic, and the partition. A good partition minimizes λ_{xc}, the flow between core nodes, and balances the total number of packets processed by each core.

We model the expected per-core cost of a single edge packet by:

$$E[cost] \quad = \quad E[H] * hc + ppc + E[X] * ppc$$

This equation indicates the expected amount of work of one edge packet. Here H is the expected path length of a packet, hc is the CPU cost of a hop, ppc is the per-packet CPU cost to enter and exit a core node, and X is the expected number of cross-core transitions along a packet's path in the target topology, G_t. This equation assumes that load is balanced; each core receives λ_{edge} and λ_{xc}. Thus adding cores or increasing efficiency capitalizes on all available resources. We parameterize our model to our hardware ($hc = 0.5\mu$s, $ppc = 8\mu$s), which is described in Section 6. We assume the system is CPU limited, and calculate packets per second using this model. X depends on the quality of the partitioning and relates to efficiency as $X = \frac{1-\sigma}{\sigma}$.

We compare adding cores to a worst-case partitioning, one where each edge packet must cross to another core for each hop along its path. We set $X = H - 1$ because we assume that edge packets are sent to the core assigned the path's first edge. Solving $H - 1 = \frac{1-\sigma}{\sigma}$ for σ yields a lower bound on efficiency for partitioning, $\Omega(1/H)$.

Figure 4 shows the expected emulation capacity for this system when the average path length is 10. The worst-case efficiency is constant at $1/10$. Each straight line represents emulation capacity in packets per second (PPS) when scaling purely by increasing the number of core nodes. Though the scaling is linear, even a 4 core system does not have the emulation capacity of a single core (75K PPS) .

In contrast, the curved lines are 2 and 4 core configurations with increasing efficiency. To match single-core performance we either need to increase efficiency to 37% when using 2 cores or to only 16% when using 4 cores. This graph illustrates that a good partition is crucial to minimizing the incremental cost of scaling distributed network emulation. Note that efficiency must remain constant going from a k-way partition to a $k+1$-way partition to achieve a linear gain in emulation capacity.

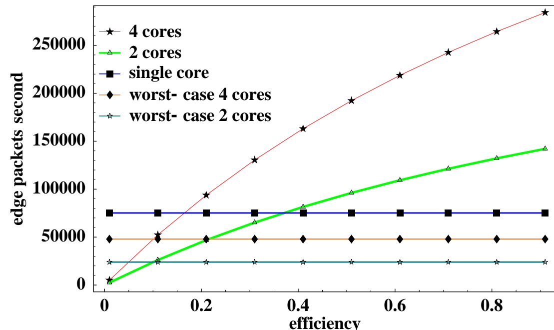


Figure 4. One can scale emulation capacity by increasing the number of cores or creating a partition with higher efficiency, σ. The 4-core worst-case partitioning (whose efficiency is constant at 10%) provides lower λ_{edge} than a single core, because packets take many cross-core hops.

5.2 Graph Partitioning Algorithms

We use two successively sophisticated graph partitioning schemes: k-cluster and METIS [6]. k-cluster partitions G_t into k connected components of equal size by iteratively building connected components through breadth-first searches. k-cluster is not novel; variants have previously been explored [1]. The shortcoming of this approach is that edges are treated equally, when in fact communication patterns favor some edges and routes more than others.

More sophisticated partitioners take inter-partition flows into account. We use the multi-level graph bisection algorithms in METIS [6], a graph partitioning library. This package uses heavy-edge matching to coarsen the graph, a greedy graph growing partitioner (a partitioning algorithm that grows connected components like k-cluster but considers flow contributions of added edges), and uses a modified Kernighan-Lin (KL) [7, 3] to refine the graph. We considered a sophisticated approach called spectral partitioning [9], however our initial experiments indicated that partitions were no better than METIS and often took longer to run. We have verified that METIS calculates the optimal solution for small graphs we solved through brute force (up to 90 vertices).

5.3 Flow Graph Construction and Static Weighting

Multi-level graph partitioners, such as METIS, require a vertex-partitionable graph with edge and vertex weights.

First we develop a simple transform on $G_t(V_t, E_t)$ to create a *flow graph* $G_f(V_f, E_f, W_{Vf}, W_{Ef})$ whose edge weights reflect edge-to-edge packet flows in the target topology. Vertex partitions of $G_f(V_f, E_f, W_{Vf}, W_{Ef})$ map surjectively onto edge partitions of $G_t(V_t, E_t)$. Next we describe an algorithm which weights the flow graph, using properties of the edges in G_t to provide an initial partition without knowledge of the communication pattern.

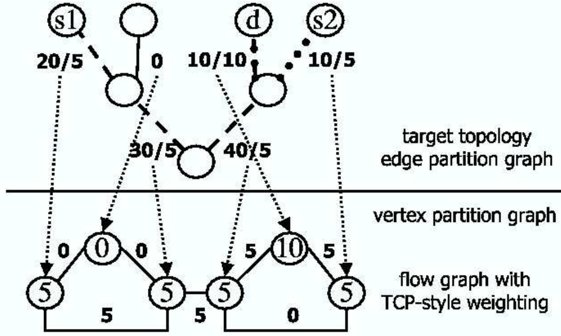


Figure 5. We transform the target topology into a fbw graph, G_f, in which a vertex partition determines the edge assignment. In this example, edges in G_t are annotated with (capacity,use). The weights of the edges of the fbw graph are computed from observing two fbws of weight 5 from $s1$ and $s2$ to d.

First we create a vertex v_i in flow graph G_f for each edge e_i in G_t. If two edges e_1 and e_2 in G_t are incident on the same vertex, then we create an edge in G_f between the two vertices v_1 and v_2 in G_f. The edge weights W_{Ef} represent the flow between two edges in G_t. The vertex weights, W_{Vf}, are the flow through an edge in G_t and represent the cost of emulating pipe v. This transform creates a flow graph G_f with $|E_t|$ vertices. If the maximum degree of vertices in the emulated topology graph is d, then the number of edges in G_f remain bounded by a constant factor $O(|E_t| * 2 * d)$.

The quality of the partition depends directly on the flow graph edge weights. Figure 5 shows the results of transforming the topology into a vertex-partionable graph and weighting the edges and vertices according to an observed communication pattern. Initial vertex partitions require a weighted flow graph, but the communication patterns are unknown until the emulation (and application) is run. We develop a simple algorithm to weight the flow graph based on static link capacities in the original target topology, G_t. This weighting assumes maximizing, congestion responsive flows, and an all-to-all communication between all clients in the target topology. The resulting flow graph weights are conservative; they approximate future network conditions. Section 5.4 explores deriving edge weights dynamically during an emulation experiment.

```
For each client:
    Find shortest paths to all clients
    PF = 0 for each edge in this tree
    For each path:
        Find the minimum bandwidth on path
        For each edge e in path
            PF[e] = MAX(minimum bandwidth, PF[e])
    For each path:
        Consider consecutive edges e1 and e2
        fbwmap[e1][e2] +=
            MIN(PF[e2], BW[e2] - fbwmap[e1][e2])
```

Figure 6. Pseudocode for fbwgraph weighting. Each edge in G_t is annotated with a partial fbw variable PF_E.

Figure 6 illustrates the conservative weighting algorithm in detail. The result of this algorithm is an estimate of the amount of flow each edge contributes to its neighbor edges, the *flowmap*. The algorithm takes as input the target topology G_t whose edges have link capacity, BW. The output of this algorithm is a set of flow graph edge weights W_{Ef} for G_f; it is stored as an array of binary trees indexed by source edge and destination edge in G_t. Once the flowmap has been created, flow graph vertex weights are calculated as the sum of the incident edge weights.

The intuition behind this algorithm is that all communicating pairs create a maximizing TCP-like flow which attains throughput equal to the bottleneck link between source and destination. The flow on each link of that path is set to be at least this bottleneck bandwidth. The contribution of flow from one link to a successor link (the edge weights in the flow graph) is the flow found across the successive link unless this contribution exceeds the capacity of the successive link. This algorithm creates a flowmap that in essence is worst case.

While a flow from edge $e_{j,k}$ into edge $e_{k,l}$ cannot contribute more than the link capacity of $e_{k,l}$, the sum flow into $e_{k,l}$ from all edges $e_{*,k}$ may exceed $BW_{k,l}$. However, the point of this weighting is to give the partitioner a worst-case view of making a cut across any single edge. Looping through all source-destination pairs takes $O(N^2)$ time, the number of client vertices in G_t. We calculate the shortest-path tree each time through the inner loop, taking $O(|E|$ log $|V|)$ time. We traverse each edge in the tree a constant number of times, taking $O(|E|)$ time, and at each edge we update the flowmap, taking $O(|E|$ log $|E|)$ time. The total runtime is $O(N^2 * |E|$ log $|E||V|)$ time.

5.4 Dynamic Partitioning

The previous section showed how we can create a flowmap that is a conservative estimate of the flows across

the target topology. There are two key differences between this weighting and the real flows from the application. First not all paths may be used and not all flows may be maximizing. The actual communication pattern may present the partitioner with flows that can be contained on a fewer number of core nodes. Second many network systems adapt communication to network conditions, resource contention, or object availability (caching). Dynamic communication patterns can unbalance load across the core nodes, and create an inaccurate emulation when a core becomes overloaded (either at the network or CPU). Thus, there are two objectives of dynamically partitioning the topology during an emulation: optimize the partition for the communication pattern, and reduce periods of inaccuracy. Dynamic partitioning requires: i) observation of the communication patterns, ii) recomputing the flowmap based on the observation, iii) calculating the new partition, and iv) distributing the new assignment of edges to core nodes.

The system performs these four tasks every epoch, a configurable time period. Short epochs contain an emulation that is drifting into overload, but long epochs amortize the cost of reassigning pipes across the core nodes. We minimize the overhead of pipe assignment changes by matching each new partition to the core which has the largest overlap with its current set of pipes. Pipes must be reassigned carefully during an emulation to maintain accuracy. For example, if two cores emulate the same pipe concurrently, the bandwidth may double through that link in the target topology.

To orchestrate the assignment of pipes we divide core nodes into two groups: those aware of the new pipe owner, and those that are not. Ownership changes occur in four steps: i) update the new owning core, but do not emulate the pipe, ii.) update the old owning core, iii.) tell the new owner to start emulating the pipe, and iv.) update the rest of the cores. Step one alerts the new owner to begin emulating the pipe if traffic for that pipe arrives from the old owner. In step two the old owner begins to forward traffic to the new owner. Note that until step four completes, some traffic takes an extra hop through the old owner.

6 Evaluation

We implemented static and dynamic partitioning in the ModelNet network emulator [14]; a review of the ModelNet system can be found in Appendix A. This section compares random, k-cluster, and METIS partitioners on two types of static network topologies. The first is a configurable toy topology and the second is an Internet-like topology generated by iNet [16]. Initial dynamic partitioning experiments indicate that repartitioning yields important improvements in efficiency during experiments.

Unless otherwise noted, all experiments in this paper run on the following hardware. Edge and core nodes are 1.4Ghz Pentium IIIs with 1GB of main memory running Linux version 2.4.17 and FreeBSD-4.7-STABLE respectively. They are all connected via 1Gb/s Ethernet to a Cisco Catalyst 4000 gigabit switch. Per-packet hop emulation cost is 0.5μs, and the per-packet cost to enter a core is 8μs. A single cross-core hop adds 100μsto the latency of an emulated path. All partitioning experiments use payload caching. Network traffic consists of maximizing TCP flows generated with user-level microbenchmark applications. The iNet topology is a 4000 node graph with 200 stubs and 400 clients. All static partitioning tests have 400 clients over 8 edge machines.

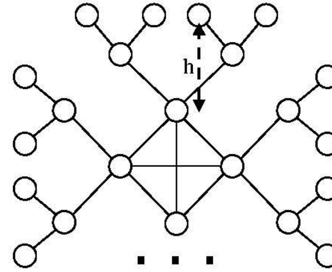


Figure 7. Target topology used in partitioning tests. Paths between trees have length $2h + 1$, where h is tree height.

Typical scalability studies that show increasing system capacity (larger problem sizes) scaling with increasing hardware resources are difficult to apply to scaling network emulation. The key problem is that the notion of problem size is ambiguous due to the NP-hard nature of the partitioning problem. Any result is dependent on traffic intensity, communication pattern, the number of cores, and the structure and link characterstics of the network topology. For example, showing an increase in emulation capacity in packets per second would not show the ability of the system to handle topologies of increasing diameter or different,dynamic communication patterns. In this paper we focus on showing improvements in emulation efficiency given a constant problem size. We also employ a toy topology whose complexity can be increased in a controlled manner to evaluate the impact on the partitioning scheme.

The toy target topology is an n-ary tree with a well-provisioned core and slow last-mile client links. Figure 7 shows the topology. The root of the tree can have arbitrary degree, allowing us to hang many n-ary trees from the core of the topology in a fully connected mesh. In our experiments, 4 trees hanging off of the central mesh. This topology guarantees the path length between clients in separate trees will be $2*h+1$, where h is the height of the tree. Thus by increasing h, we increase the diameter and the number of links shared by multiple routes.

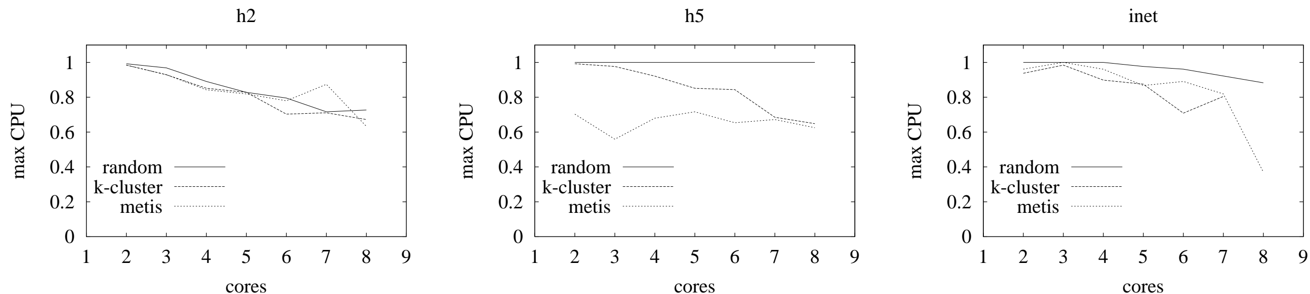


Figure 8. The maximum CPU utilization during any test for the three partitioning techniques. Note that no partition is guaranteed to be accurate with h2 and two cores. With random partitioning, the h5 graph causes a core to be overloaded (and the emulation to potentially be inaccurate) no matter the cluster size.

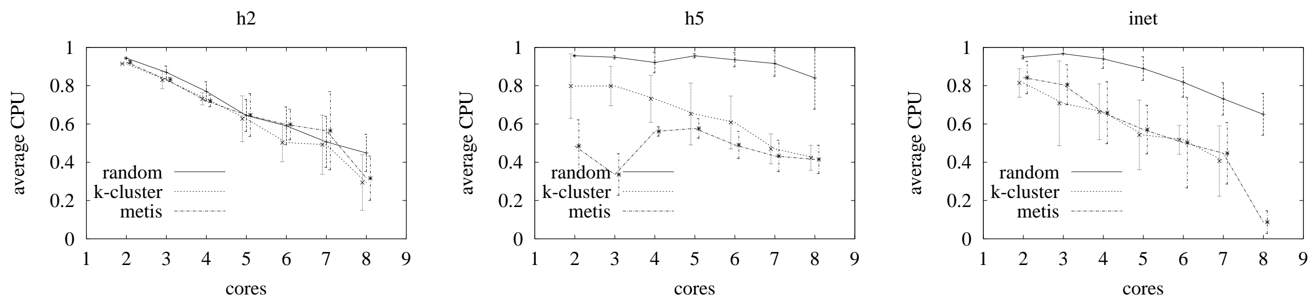


Figure 9. Average CPU utilization with standard deviation error bars (staggered horizontally so they do not overlap). METIS and k-cluster consistently outperform random on more complicated graphs: h5 and inet.

6.1 Static Partitioning

We compare three partitioning strategies (random, k-cluster, and METIS) using three graphs, $h = 2$ (h2), $h = 5$ (h5), and an iNet topology. We run k-cluster five times and use the best partition. The graphs use the congestion responsive worst-case weighting described in Section 5.3. In these experiments each client connects to four random clients (in separate trees) in the topology. The topology and traffic pattern is held constant as we increase from 2 to 8 cores. The workload consists of 400 TCP streams whose initial pipes are constrained to 3Mb/s. This generates approximately 150K pkts/sec and overloads a single core.

Figure 8 shows the CPU utilization at the most loaded core for each topology and partitioning strategy. No partitioning strategy is able provide emulation without an overloaded core for the h2 topology for a 2 core configuration. The h5 topology increases the complexity, diameter, and number of routes sharing links. In this case no random partition, even with 8 cores, can reduce the highest CPU load experienced by a core. In contrast, METIS keeps the load on all cores beneath 71%. The Internet-like topology seems to quantitatively be a mix of the two toy topologies. 8 core random partitions keep the load below 100%, while METIS and k-cluster can provide accurate emulations with 4 cores. Figure 9 shows the average CPU utilizations across all cores in each experiment (with standard deviation error bars). In general, METIS and k-cluster appear to trade some load balancing for a decrease in CPU utilization.

Figure 10 shows the efficiency of each partitioning scheme across the three topologies. Recall that efficiency is the ratio of edge to total traffic of the system. The small diameter and greater fraction of edges shared by few routes makes the simple h2 topology amenable to even random partitioning. METIS more than doubles efficiency on the h5 topology with 4 cores relative to a random partitioning (36% improvement compared to k-cluster). Comparing this result to the h5 graph in Figure 9, we can see that it translates into approximately halving the CPU utilization (equivalently doubling the emulation capacity). However note that after that point efficiency does not improve, and the system appears to be able to scale linearly, at a factor proportional to the efficiency, at least to 8 nodes. For iNet, both METIS and k-cluster results are similar to h5. Thus for topologies with larger diameters or Internet-like structure smart partitioning significantly improves emulation capacity.

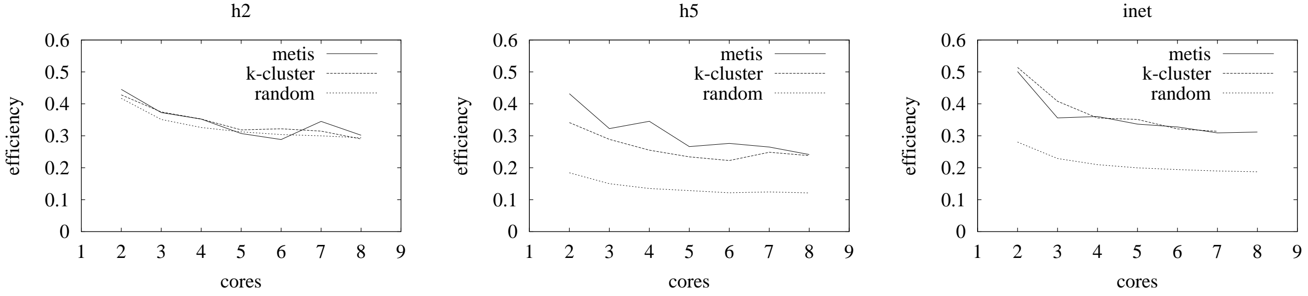


Figure 10. These graphs show emulation efficiency for $h = 2$, $h = 5$ and an iNet topologies for random, k-cluster, and METIS partitioning. We vary the number of cores from 2 to 7. For topologies with larger diameters ($h = 5$ and iNet), a smart partitioning scheme yields large benefits in efficiency. In the case of iNet, METIS provides 65% higher efficiency than random with an 8-way partition. Further efficiency decreases slowly as the number of partitions is increased. This means that we can extract linear scaling from additional core nodes.

A constant efficiency indicates that emulation capacity will scale linearly with increasing core nodes, but only if the load is balanced. This highlights the tradeoff that the graph partitioning algorithms must make. Though an increase in the number of cores provides more resources, the number of pipes to assign remains constant. Thus the partitioner can decide to spread the load evenly across the cores, potentially increasing the total cost of the emulation, or the partitioner can leave some nodes underutilized and maintain efficiency. Figure 9 shows that average CPU utilization consistently improves with increasing cores, but unbalanced load is likely to cause at least one core to be overloaded. If we look at the iNet topology in Figure 10 and Figure 9, we see that load is increasingly unbalanced as the number of cores increases. This indicates that graphs with power law connectivity distributions will see a decreasing benefit to scaling with additional cores even with intelligent partitioners.

We were surprised at the performance of k-cluster relative to METIS. However the uniform traffic pattern of these experiments does not stress the weakness of k-cluster; it does not use flows when determining a partition. For that reason and the fact that METIS performed as well or better in all cases, we use METIS in our dynamic partitioning scheme.

6.2 Dynamic Partitioning

Recall the two goals of dynamic partitioning: optimize the partition for the communication pattern, and reduce the possibility of inaccuracy due to shifts in communication pattern. Dynamic experiments run for 300 seconds and we vary the communication pattern to stress the initial static assignment. We use the h2 toy topology. First two target subtrees are chosen. Clients not in either of those trees pick two

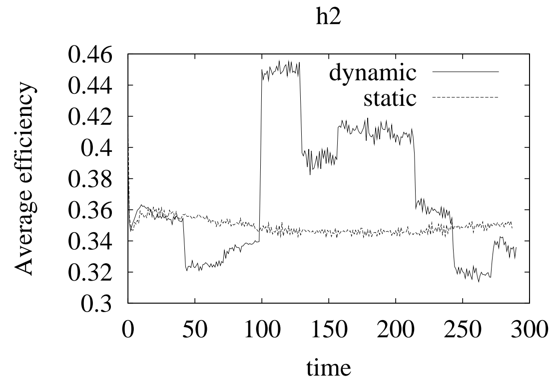


Figure 11. Dynamic partitioning efficiency versus a static partition for a 300 second experiment using the h2 topology. Repartitioning during this experiment maintains higher efficiency, increasing emulation capacity to handle changes in load and communication pattern.

random targets, one each from the two target sub-trees. At a random time, each client changes targets from the initial tree to the second. We set our epoch length to 30 seconds.

Figure 11 plots the efficiency of the initial static partitioning against the efficiency of the dynamic scheme during the 300 seconds. Note that because our dynamic communication pattern is essentially uniform (each target tree is on average servicing the same number of flows), static efficiency only bows slightly in the middle of the graph as clients move between targets. Repartitioning in the dynamic scheme maintains higher efficiency throughout the majority of the experiment. This indicates that observing real communication patterns creates significantly better partitions than a static weighting. The lower efficiency at the beginning and end of the experiment is due to a current limitation in ModelNet that prevents us from reassigning entry

pipes (pipes that connect clients to the rest of the topology). We are currently removing this limitation. Epochs can be as short as 15 seconds; the bottleneck is recomputing the flowmap from the last observations.

7 Conclusion

This paper develops and evaluates static and dynamic topology partitioning techniques for distributing network emulation across multiple processors. We show that a simple technique, payload caching, reduces the link stress on the emulator's underlying physical network. We compare and contrast two partitioning algorithms to random partitioning. Further we develop methods to map the target topology into a graph that is vertex partionable and to weight this graph according only to the bandwidth attributes of its links. We design a dynamic partitioning scheme to meet the challenges of emerging large-scale adaptive applications (peer-to-peer, overlays, etc.,). These techniques were implemented and empirically evaluated in a large-scale network emulator, ModelNet. For a given number of processors, our results indicate that static graph partitioning can double emulation efficiency versus a random assignment for Internet-like network topologies. Finally, our dynamic partitioning scheme shows that observations of real communication patterns improve the partitioning and emulation efficiency. The generality of these techniques make them equally valuable to network simulation.

References

[1] P. Ciarlet, Jr and F. Lamour. On the validity of a front oriented approach to partitioning large sparse graphs with a connectivity constraint. Technical Report 94-37, Computer Science Department, UCLA, Los Angeles, CA, 1994.

[2] J. Cowie and H. Liu. Towards realistic million-node Internet simulations. In *Proceedings of the 1999 International Conference on Parallel and Distributed Processing Techniques and Applications*, 1999.

[3] D. M. Fiduccia and R. M. Mattheyses. A linear time heuristic for improving network partitions. In *Proceedings of 19th IEEE Design Automation Conference*, pages 175–181, 1982.

[4] M. Garey and D. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman, 1989.

[5] P. Huang, D. Estrin, and J. Heidemann. Enabling large-scale simulations: selective abstraction approach to the study of multicast protocols. In *Proceedings of the International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems*, pages 241–248, Montreal, Canada, July 1998. IEEE.

[6] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal of Scientific Computing*, 20(1):359–392, 1998.

[7] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *Bell Systems Technical Journal*, 49(2):291–308, 1970.

[8] R. Pan, B. Prabhakar, K. Psounis, and D. Wischik. SHRiNK: A method for scalable performance prediction and efficient network simulation. In *Proceedings of IEEE Infocom*, March 2003.

[9] A. Pothen, H. D. Simon, and K. P. Liou. Partitioning sparse matrices with eigenvectors of graphs. *SIAM J. Matrix Anal. Appl.*, 11:430–452, 1990.

[10] D. M. Rao and P. A. Wilsey. Simulation of ultra-large communication networks. In *Proceedings of Seventh International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunications Systems*, 1999.

[11] G. F. Riley, R. Fujimoto, and M. H. Ammar. A generic framework for parallelization of network simulations. In *MASCOTS*, pages 128–, 1999.

[12] R. Simmonds, R. Bradford, and B. Unger. Applying parallel discrete event simulation to network emulation. In *The 14th Workshop on Parallel and Distributed Simulation (PADS 2000)*, May 2000.

[13] B. K. Szymanski, A. Saifee, A. Sastry, Y. Liu, and K. Madnani. Genesis: a system for large-scale parallel network simulation. In *Proceedings of 16th Workshop on Parallel and Distributed Simulation PADS2002*, Washington, DC, May 2002.

[14] A. Vahdat, K. Yocum, K. Walsh, P. Mahadevan, D. Kostic, J. Chase, and D. Becker. Scalability and accuracy in a large-scale network emulator. In *Proceedings of the Fifth Symposium on Operating System Design and Implementation (OSDI)*, December 2002.

[15] B. White, J. Lepreau, L. Stoller, R. Ricci, S. G. M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *Proceedings of the Fifth Symposium on Operating System Design and Implementation (OSDI)*, December 2002.

[16] J. Winick and S. Jamin. Inet-3.0: Internet topology generator. Technical Report CSE-TR-456-02, Computer Science Department, University of Michigan, Ann Arbor, MI, 2002.

[17] K. Yocum and J. Chase. Payload caching: High-speed data forwarding for network intermediaries. In *Proceedings of USENIX Technical Conference*, 2001.

[18] X. Zeng, R. Bagrodia, and M. Gerla. GloMoSim: A library for parallel simulation of large-scale wireless networks. In *Workshop on Parallel and Distributed Simulation*, pages 154–161, 1998.

A ModelNet

We use ModelNet to explore the scalability of network emulation. ModelNet is a network emulation environment for the evaluation of large-scale network systems [14]. It allows unmodified distributed applications to run over unmodified operating systems across emulated Internet-like topologies. The emulation environment accurately captures the effects of queueing, cross traffic, and failures. We have used ModelNet to reproduce results from both the wide area and from NS.

Network emulation allows system researchers to run unmodified network systems/applications across Internet-like topologies in the convenience of a local machine room. The physical machines are split into two sets: those that run the application (*edge nodes*) and those that will emulate the network or *target* topology (*core nodes*). Figure 12 gives a high-level view of emulation on a physical machine cluster. *Core nodes* emulate each hop of a packet's path from source to destination in the target topology. The ModelNet edge nodes host multiple *virtual nodes* (VN)s, each of which a

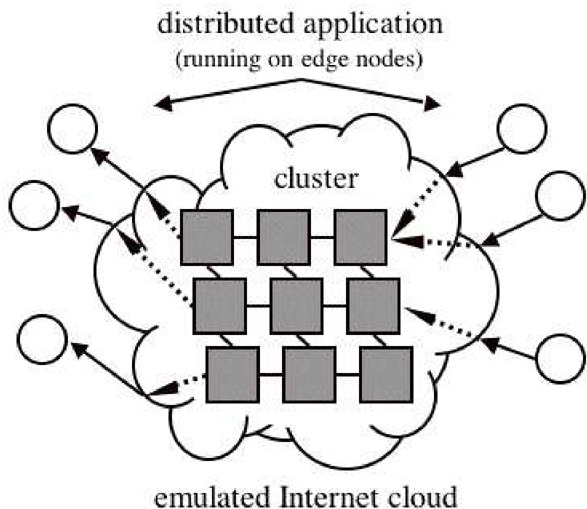


Figure 12. ModelNet consists of two types of nodes: edges and cores. Edge nodes (circles) host instances of the distributed application and are configured to send their data to the ModelNet core nodes (squares). The cores cooperate to subject the traffic to the bandwidth, latency, and loss of the target network topology.

unique IP address corresponding to its location in the emulated topology. Each edge node routes its network traffic to a set of core nodes. Packets exiting the core are delivered to the edge host hosting the destination VN. While the edge nodes are unmodified, the ModelNet core nodes run a loadable FreeBSD kernel module.

Each hop in the target topology is emulated with a queue of bounded length, which we call a *pipe*. A pipe buffers a specified maximum number of packets; overflows result in packet drops. A packet's exit time is computed for each pipe according to the latency, bandwidth, and loss rates specified by the target topology. The packet is dequeued at its exit time and moves to the next pipe or exits the emulated topology to the destination process on an edge node. Delay and congestion characteristics are accurately captured as pipes are shared among simultaneous flows.

ModelNet achieves emulation accuracy through careful coordination of kernel components. Under load, ModelNet preferentially emulates the delay for packets that have already entered the core rather than service packets waiting to enter the core through hardware interrupts. This means that core node saturation (at the CPU or network) results in dropped packets (at the physical level) rather than inaccurate emulation of packets in a pipe (note the distinction between physical drops and emulated drops in the core). By observing physical drops, inaccurate emulation runs are flagged. We can only maintain fidelity to the emulated topology when no core is overloaded. In this sense, accuracy is binary.

For a given CPU load, a ModelNet core node can emulate x hops for each packet given an arrival rate of y packets per second (PPS). As the number of hops that a packet takes inside a core increases, the throughput in packets per second drops. A single ModelNet core running on a 1.4 Ghz Pentium III can emulate approximately 120,000 packets/second traversing 3 hops, or 70,000 packets/second emulating 12 hops.

We have modified ModelNet to support the METIS partitioner and dynamic partitioning. A kernel-based logging facility, NetLog, captures events for monitoring traffic at the cores and sends them to a central log server. Dynamic partitioning/reassignment leverage user-level programs on the cores to read pipe status and modify pipe ownership. These programs use the FreeBSD `sysctl` interface to access kernel data.