Towards High Fidelity Network Emulation

Lianjie Cao, Xiangyu Bu, Sonia Fahmy, Siyuan Cao
Department of Computer Science, Purdue University, West Lafayette, IN, USA
E-mail: {cao62, xb, fahmy, cao208}@purdue.edu

***Abstract*—Instantiating a distributed application that involves extensive inter-node communication onto infrastructure is a challenging task. In this work, we focus on the special case of mapping a network emulation experiment onto a cluster comprising several (possibly heterogeneous) physical machines. We automatically profile the available physical machine resources, and use this information, together with the characteristics of the experimental topology, to determine an efficient mapping that preserves performance fidelity. We design an algorithm, which we call the *"Waterfall" algorithm*, and integrate it into a complete framework for profiling and mapping. We demonstrate the effectiveness of our framework via simulations and two sets of Crossfire Distributed Denial of Service attack testbed experiments.**

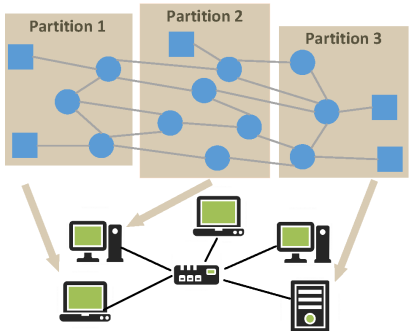


Fig. 1. Mapping onto a (possibly heterogeneous) cluster.

I. Introduction

In today's cloud computing systems, efficiently mapping a distributed/networked application onto a cluster of machines is extremely important. In this paper, we consider a special case of this problem: mapping a network emulation experiment onto a cluster of possibly heterogeneous physical machines (PMs). A key requirement for network emulation is maintaining high *performance fidelity* for any network experiment. By performance fidelity, we mean that the performance results of the experiment (*e.g.*, packet latency, packet loss and flow completion times) should match the results obtained from an experiment conducted on a physical topology identical to the experimental topology. When the experimental topology is large or the experiment is highly traffic-intensive, a physical machine (PM) running the experiment may become overloaded, leading to fidelity loss [1], [2]. A natural way to address this problem is to extend the network emulator to run across multiple PMs – often referred to as "cluster mode" [3], [4].

In this paper, we consider a cluster comprising a set of PMs connected via a powerful switch as shown in Fig. 1. The PMs may include various types of hardware, possibly purchased at different times. For example, the GENI [5], DETER [6], [7], and Emulab [8] clusters each have many types of machines, since upgrades cannot be performed for the entire cluster at the same time. Heterogeneity is likely to increase with the proliferation of testbeds and cloud environments with different virtualization technologies. Taking full advantage of each PM in the cluster while maintaining high performance fidelity for the experiment is a key challenge.

To address this challenge, we design a complete framework for mapping a network experiment onto a (possibly heterogeneous) cluster. We leverage MaxiNet [9], which extends the popular Mininet network emulator [4], [3] to run on a set of physical machines. Our framework quantifies the capacity of the PMs in the physical cluster, and uses this information to map an input experimental topology onto some or all of the cluster PMs. We design an algorithm, *the Waterfall algorithm*, that leverages a popular graph partitioning algorithm (we choose METIS [10], [11] in this paper), and we plug it into our framework.

Our design is guided by the following principles:
(1) Integrity and Fidelity: The network components of the original experiment (hosts, switches, routers, or links) must all be correctly mapped. Fidelity includes two aspects: (i) not overloading any machine and (ii) not overloading the links between those machines.
(2) Best effort: Performance fidelity is preserved when possible. Mapping is completed, however, even when the resources required by the experiment exceed available PM resources. In this case, a best effort experiment can be conducted. The user is warned about potential fidelity loss and can employ techniques like time dilation [12], [13] or add more resources to the cluster.
(3) Judicious use of resources: When there are sufficient PMs to accommodate the experiment, we use as few PMs as possible without jeopardizing fidelity. This allows more than one user to utilize the cluster at the same time when possible.

The key contributions of this work include: (1) We introduce a *capacity function* for each PM that models its traffic processing capacity as a function of its CPU share. The capacity is compared to the resource requirements of the experimental topology during the mapping process; (2) We design the *Waterfall* algorithm, which iteratively invokes a graph partitioning algorithm [10], as a plugin for our framework. The Waterfall algorithm partitions and maps experimental nodes onto possibly heterogeneous PMs, reducing inter-PM traffic. The algorithm preserves the fidelity when there are sufficient resources, but uses only a subset of the physical machines when possible (which a direct application of graph partitioning cannot accomplish); and (3) We evaluate our framework via simulations and testbed experiments with Crossfire distributed denial of service (DDoS) attacks, and demonstrate that it achieves higher fidelity than approaches that are agnostic to

cluster or experiment characteristics, while efficiently using cluster machines.

The remainder of the paper is organized as follows. Section II summaries related work. Section III describes the design of our framework and our Waterfall algorithm. We evaluate our framework in Sections IV and V, and conclude in Section VI.

II. Related work

Mapping a network experiment onto available PMs is a classic resource allocation problem. A network experiment can be modeled as a weighted graph representing the network topology and (potentially) information about network traffic flows. Hence, the mapping problem can be translated into a graph partitioning problem that attempts to minimize the edge cut weights.

Graph partitioning is known to be NP-hard and has been studied for decades. The Kernighan-Lin (KL) algorithm [14] is one of the earliest effective heuristics to address 2-way partitioning by swapping vertices in the two initial partitions to achieve a smaller edge-cut. Spectral algorithms [15], [16] partition a graph by computing the eigenvectors of the adjacency matrix of the graph.

The multilevel approach proposed in [10], [11], [17], [18] constructs a sequence of increasingly coarsened graphs and partitions the smallest graph. It eventually uncoarsens the partitioning results back to the original graph. Heuristics are used in each phase to increase the partitioning quality. These algorithms take as input the number of partitions and the *shares* of each partition. We leverage single-objective, multi-constraint METIS [10], [11] in this paper, and focus on balancing the seemingly-conflicting goals of performance fidelity and judicious use of resources.

Another related line of work is work on service placement in data centers. The Virtual Machine (VM) placement problem [19], [20], [21] allocates VMs to reduce traffic/latency and achieve high PM utilization. The requests do not typically include network devices, such as switches and routers, though. The virtual network embedding problem [22], [23] determines a solution to map each virtual network to a substrate network, addressing potentially different performance objectives (*e.g.*, latency or throughput). While this is similar to our problem in that the underlying resources to be allocated include physical network infrastructure (switches, routers and links), our work considers an additional factor: the traffic processing capacity of a physical machine.

Testbed mapping [24], [7], [5], [1] instantiates a testbed user's experiment onto physical resources. This includes both one-to-one mapping (one experimental or virtual node (vnode) to one physical machine (PM)) and many-to-one mapping (multiple vnodes to one PM) [25]. Once users "swap in" their experiments, they usually last for a long time [24], [7]. Historical records can be used to improve the allocation solution (as in DETER `assign+` [7]). Flow-based scenario partitioning (FSP) [26] partitions a network experiment based on traffic flow information and combines the results from partitions. EasyScale [1] is a framework for leveraging multiple scaling techniques. Our work is complementary to this work, and can be used to map an individual EasyScale sub-topology, for example.

Virtualized Emulab [27] is closely related to our work. It extends the `assign` algorithm [24] used in Emulab with heuristics to enable large scale network emulation. It does not, however, develop a unified resource capacity abstraction for physical nodes and the experimental topology. Virtualized Emulab divides a physical node into a number of slots and asks the user to provide information on how many slots an experimental (virtual) node needs. The capacity of the "loopback" device that carries traffic among virtual nodes on the same physical node is not quantified. VT-Mininet [28] proposes an adaptive virtual time system [12], [13] to scale Mininet [4], [3]. Time dilation [12], [13] prolongs the experiment duration, and can be used in conjunction with our work in cases when resources are insufficient.

Finally, MaxiNet [9] develops a framework for cluster-mode Mininet. By default, MaxiNet employs the METIS graph partitioning software [11] to split a network topology into a given number of partitions (sets). Each partition can then run on a PM in the cluster. MaxiNet with METIS does not profile the physical machines or consider CPU requirements of experimental nodes, which we focus on in this paper. Mininet 2.2.0 also introduced an experimental cluster mode. The cluster mode provides several simple placement algorithms (`SwitchBinPlacer`, `RandomPlacer`, `RoundRobinPlacer`), none of which addresses fidelity concerns.

III. Resource Allocation Framework

Current network emulators use virtualization technologies such as virtual machines, containers, and software switches to emulate a network. Performance *fidelity loss* occurs when there are insufficient physical resources (*e.g.*, CPU and memory) to process emulation events. For instance, fidelity loss may occur when a large number of flows are transmitting at the same time from different end hosts.

Our framework considers the experiment mapping problem from a resource allocation perspective. We design and implement new modules that take the network experiment description from the network emulator, partition the experiment, and allocate physical resources to each partition. Any cluster of available machines – which may not all be identical [7], [5] – can be used to transparently conduct network experiments using our software and a distributed emulator. Our new modules, which together constitute a resource allocation middle layer, interface with MaxiNet [9], which extends the popular Mininet-HiFi emulator [3] to cluster mode.

The key questions in devising this mapping and resource allocation framework include: (1) How do we quantify the physical resources available in the (possibly heterogeneous) cluster? (2) How do we partition a network experiment to match the available underlying physical resources to achieve high performance fidelity? and (3) How do we reduce the number of allocated PMs, to allow other users to also utilize the cluster when possible?

A. Resource Quantification

In the context of network emulation, preserving performance fidelity means assuring users not only correct connectivity of their topology, but also accurate performance (*e.g.*,

link bandwidth, delay, and loss rate) of switches and end hosts. For software switches (which handle network traffic in network emulators), in particular, performance is limited by their implementation and available physical resources. Performance fidelity is degraded when the software switches are overwhelmed by network traffic in an experiment. Therefore, we need to quantify both the hardware specifications such as CPU type and memory size, and the traffic processing capacity of software switches for each PM. While a quantitative representation of physical resources is necessary to optimally map experimental nodes onto heterogeneous PMs, to the best of our knowledge, no systematic approach to do so exists today. In our framework, we model a PM by the properties in Table I.

TABLE I. PROPERTIES OF PM i

Property	Description
θ^i	A multiplier to normalize the single core performance of CPU.
U^i	Maximum number of CPU shares available.
U^i_{smin}	Minimum number of CPU shares for packet processing.
U^i_{smax}	Maximum number of CPU shares for packet processing.
u^i_s	Number of CPU shares for packet processing.
u^i_h	Number of CPU shares for emulated hosts.
$C^i(u^i_s)$	Capacity function of the PM with domain $[U^i_{\text{smin}}, U^i_{\text{smax}}]$.

1) Modeling CPU Performance: Several factors affect the performance of modern CPUs. Our PM model includes the two dominating factors for modeling CPU performance: single core performance and number of cores. Each core offers 100 CPU shares; therefore, U^i is usually set to (100 $\times$ total cores), but can be set to a smaller value to take system overhead into consideration (*e.g.*, reserve 10 shares for OS overhead).

Since PMs may be heterogeneous, the model must also take the difference in single core performance into account, which is reflected by θ. θ reflects the relative strength of single core performance. For instance, if $\theta^i = 1$ and $\theta^j = m$ for PM i and PM j, then the single core performance of PM j is m times as fast as that of PM i. This property can be set using results from benchmarking tools, but using CPU frequency suffices in many cases.

While certain factors that affect CPU performance, such as SMT, can be modeled by the properties we use, other factors like dynamic frequency scaling cannot. We assume that due to the nature of network experiments, all cores are heavily utilized, so θ^i reflects single core performance when *all* cores of the PM are actively utilized.

Our model does not take the difference in memory capacity limit of each PM into consideration mainly because network experiments are not typically memory-intensive. However, the model can be easily extended to support memory limits by adding a memory capacity property for each PM, specifying the memory requirements of switches and end hosts in the experimental topology and using this as an additional partitioning constraint.

2) Modeling Packet Processing Capability: We determine the traffic processing capacity by running a set of experiments on each type of PM. This is challenging because (1) implementations of different software switches exhibit significantly different performance and resource usage on the same machine (*e.g.*, Open vSwitch (OVS) [29], Indigo Virtual Switch (IVS), or the Stanford reference switch (UserSwitch)), (2) throughput varies based on packet size, type of traffic, and switch instances in the network topology to be emulated, and (3) different hardware features may lead to different packet processing performance (*e.g.*, SR-IOV and number of queues on a NIC). We do not compute the impact of different software switch implementations or hardware features because the goal of our resource quantification module is to characterize the relationship between throughput of software switches and resource utilization for a given PM and software switch. If hardware is upgraded or a new software switch is introduced, our capacity functions need to be updated.

Our measurements show that most software switches are CPU-intensive. Therefore, we focus on CPU utilization in this paper and leave other types of resource limits for future work. We compute the packet processing capacity function $P^i(u^i_s)$ of PM i in packets per second, $U^i_{\text{smin}} \leq u^i_s \leq U^i_{\text{smax}}$. For instance, $P^i(50)$ denotes the maximum traffic rate that PM i can handle when allocating 50 CPU shares to packet processing.

The motivation for quantifying the traffic processing ability of a PM using a capacity *function* rather than a single value representing the maximum processing capacity is that, in network emulation (or any distributed task), a user may run on end hosts custom programs (running in containers in the case of Mininet/MaxiNet) that compete for resources with software switches. Some network emulators provide interfaces for a user to set an upper bound on how much CPU they prefer to use for each end host. For instance, a user can use `CPULimitedHost` in Mininet to limit the CPU usage of an end host. Therefore, abstracting packet processing capability to a single value is insufficient, and more dynamic representations such as capacity functions are more desirable.

Capacity functions are determined by running our resource quantification module on a simple linear topology as illustrated in Fig. 2. When measuring capacity versus CPU usage, we run the traffic generator and receiver on adjacent PMs to avoid including the CPU usage of the traffic generator in our function. Each experiment is repeated 10 times.

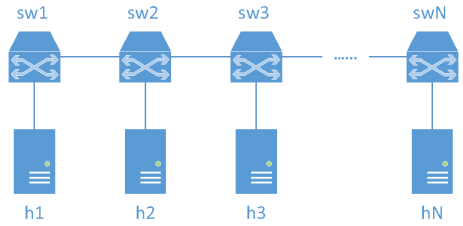


Fig. 2. Linear topology used for resource quantification

We investigated six packet sizes: 64 Bytes, 128 Bytes, 256 Bytes, 512 Bytes, 1024 Bytes, and 1250 Bytes, and observed that throughput in Mbps varies significantly at the same CPU usage for certain software switches (*e.g.*, UserSwitch), while throughput in packets per second remains stable. We therefore compute the capacity function in packets per second. Since many network emulators (*e.g.*, Mininet) and testbeds require users to specify bandwidth in bits per second, we design our partitioning module to accept both unit systems, but require hints from the users to convert capacity functions to Mbps.

During each test, we measure the reception rate (`Rx`)

and transmission rate (`Tx`) of all switch instances and their CPU utilization. Fig. 3 shows the results for 6 PMs (2 PMs are of the same model) in a cluster with dual-core CPUs and quad-core CPUs, 1.20 GHz to 2.40 GHz frequency and 4 GB to 16 GB RAM, on 4 linear topology sizes, running UserSwitch. Tests with different numbers of switches yield similar results. The relationship between CPU usage and total PM throughput is close to linear. With more cores, this relationship becomes less linear when CPU utilization increases. When CPU usage exceeds $90\% \times U^i$, the throughput becomes unstable. Therefore, we discard data at more than $90\% \times U^i$. Correspondingly, we limit the maximum number of CPU shares allocated to software switches to U^i_{smax} in our algorithm in Section III-C. Users can also set the minimum number of CPU shares for packet processing, U^i_{smin}. We perform up to fifth-order polynomial regression on the dataset we collected and select the model with least mean squared error (MSE) using 5-fold cross validation. A user can choose a different regression model, if desired. The four different traffic processing capacity functions we derived are as follows, where $U^i_{\text{smin}} \leq u \leq U^i_{\text{smax}}$ (the bounds may vary for different PMs).

$$\begin{aligned} P_{2core@1.20GHz}(u) &= 0.0168u^2 + 192.944u - 286.828 \\ P_{2core@2.39GHz}(u) &= 0.425u^2 + 285.166u - 2709.699 \\ P_{4core@1.20GHz}(u) &= 0.359u^2 + 112.275u + 4061.292 \\ P_{4core@2.39GHz}(u) &= 0.279u^2 + 316.796u + 948.393 \end{aligned} \quad (1)$$

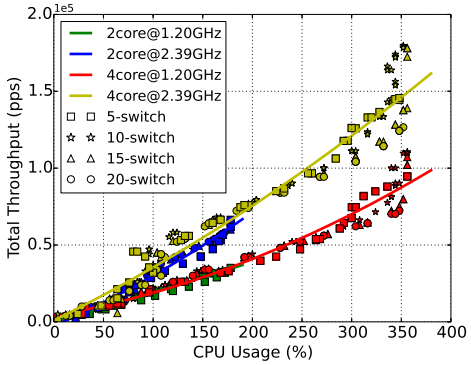


Fig. 3. Traffic processing capacity functions of the four types of PMs in our testbed

3) Property Scaling: To make the CPU shares of different types of PMs comparable, our framework scales PM properties by multiplying every CPU share-related property by θ^i. For example, the scaled U^i is $\theta^i \times U^i$, and the scaled U^i_{smax} is $\theta^i \times U^i_{\text{smax}}$. The capacity function is also scaled accordingly. For simplicity, in the sections that follow, we assume that the values have already been scaled (unless otherwise noted).

B. Topology Abstraction

We use a preprocessing module to export the network experiment information from a network emulator to external files and then to import experiment partition information back into the network emulator. This design minimizes modifications when applying our resource management layer to different network emulators.

The experiment topology (input virtual topology) is abstracted to a weighted undirected graph as follows:

1) Each end host is considered together with its adjacent switch (for simplicity, we assume single-homed hosts for now). Switches/routers and links in network topology correspond to vertices and edges in the graph, respectively.
2) *The weight of an edge* in the graph is a positive number assigned according the bandwidth (in Mbps) of the corresponding link in the network topology.
3) *The weight of a vertex* is the sum of the bandwidths (in Mbps) of the links incident onto that switch or router in the network topology. Host to switch (or host to router) links are thus considered in the weight of a vertex, not the weight of an edge.

The intuition behind step 1) above is that we need to avoid cutting links between end hosts and switches/routers when partitioning, because if we map a host and its adjacent switch onto different PMs, then we would still need to create a virtual switch for the host on its PM to connect it to that adjacent switch. MaxiNet and the default placement algorithm in Mininet cluster mode (`SwitchBinPlacer`) adopt the same approach.

Since our topology abstraction does not take traffic flow information as input, resource consumption of a switch or router must be estimated in a conservative manner. The weights assigned to edges and vertices in steps 2) and 3) above model the required processing capacity of that edge or vertex. Our current model uses link bandwidth, which impacts CPU requirements, but other models are also possible. For example, link delay affects memory requirements.

To summarize, the weighted graph $G = (V, E)$ – derived from the experimental (virtual) topology – includes $|V|$ vertices which equals the number of switches and routers, and $|E|$ links which equals the number of edges among switches and/or routers. Weights $w(v)$ and $w((a, b))$ denote the weight of vertex v, and the weight of the edge between vertex a and vertex b, respectively.

The total weight of graph $G = (V, E)$ is defined as the sum of the weights of its vertices: $w(G) = \sum_{v \in V} w(v)$. The weight of a given subgraph is defined similarly for the vertices of the given subgraph. The total number of end hosts in the input virtual topology is m, where each host is associated/linked with a vertex (switch or router) $v \in V$.

The physical cluster contains k PMs, each with a capacity function $P^i(u)$, derived from the resource quantification module. As mentioned earlier, users give hints on packet size distributions for their experiments in order to convert the capacity function $P^i(u)$ in *pps* to a function $C^i(u)$ in Mbps. In the rest of this paper, we use Mbps for PM packet processing capacity by default. Of the U^i CPU shares PM i offers, some shares, u^i_s, are allocated to packet processing, giving packet processing capacity $C^i(u^i_s)$, and at most $U^i - u^i_s$ shares for end hosts. This division of CPU shares is an estimate of the resource competition among custom programs running on virtual end hosts and software switches.

C. Partitioning and Mapping

The partitioning and mapping module is the core of our framework. The module takes three inputs: (1) Weighted graph $G = (V, E)$; (2) Host resource requirements

(*e.g.*, CPU usage) $h_j, \forall j \in \{1, \cdots, m\}$. (3) PM models $[\theta^i,\ U^i,\ U^i_{\text{smin}},\ U^i_{\text{smax}},\ C^i(u)], \forall i \in \{1, \cdots, k\}$. The module generates k' subgraphs $S_1, S_2, \cdots, S_{k'}$, each of which corresponds to an experimental partition, where $k' \leq k$ and S_i will be executed on PM i. A partition S_i includes a subset of the vertices of the original graph, where the union of these subsets is V and the intersection is ϕ. Each link in the given topology is either part of a subgraph S_i (if both its end points belong to the same output subgraph), or connects two different subgraphs (if its endpoints belong to two different output subgraphs).

1) Objective and Constraints: As discussed in Section I, the mapping algorithm must satisfy the guiding principles: integrity, fidelity, best effort, and judicious use of resources. We use resources judiciously while maintaining high performance fidelity by: (1) localizing traffic as much as possible by mapping virtual nodes that are densely connected via high-bandwidth links onto the same PM, and (2) attempting not to overload any allocated PMs, while maximizing their utilizations.

Traffic localization is motivated by the observation that most software switches yield higher throughput than physical switches. Therefore, we aim to increase the likelihood of mapping highly connected subgraphs onto a single or few PMs, which can increase fidelity. In other words, ideally we want to:

$$\text{Minimize} \sum w((a,b)), \quad (2)$$
$$\forall (a,b) \in E; a \in S_i, b \in S_j; S_i \neq S_j; 1 \leq i, j \leq k' \leq k$$

By maximizing utilization of PMs used, we are able to leave any unneeded machines for running other tasks, taking full advantage of available physical resources in the cluster. Therefore, we aim to minimize k' where $1 \leq k' \leq k$. In other words, we aim to maximize the utilization of allocated PMs, while making sure not to over-utilize them (and reduce fidelity) if possible. If not, we produce a best effort mapping.

We design the *Waterfall algorithm* that iteratively invokes the multi-constraint, single-objective, METIS algorithm [30], [10]. We use the requirements of both emulated hosts and switches as constraints, and ask METIS to minimize the edge cut (*i.e.*, localizing traffic). In each iteration, we recompute the METIS input parameters to guide METIS towards good results. We terminate when we can no longer obtain better results.

2) Partitioning Loop: Partitioning/mapping in our context is the process of dividing a weighted input graph G – given the information of k PMs, and m end host usage shares – into k' subgraphs $S_1, S_2, \cdots, S_{k'}$, and determining which physical machine i in the cluster will be used to execute each of the k' subgraphs.

The Waterfall algorithm uses an input queue to store the inputs for each iteration, and a multi-level hash set to store information from previous iterations (input, result, and evaluation metrics). We iteratively invoke a function, *single_iteration*, on the head of the queue (which is dequeued) until the queue becomes empty. When the input queue becomes empty, the best assignment at that time is chosen. Evaluation of assignments is discussed in Section III-C4.

The function *single_iteration* takes a queue entry which includes the following components: (1) Lists of chosen PMs (PMs that will be used in this iteration) and free PMs; (2) CPU shares for processing packets and for supporting end hosts for each PM i; (3) METIS-specific parameters; and (4) a termination counter (Section III-C6). The initial input is computed by function *init_input*.

The function *single_iteration* proceeds as follows. For every PM i under consideration, we compute the capacity available for packet processing from the capacity function. Then, we normalize end host CPU shares, u^i_h, and packet processing capacities of all PMs, $C^i(u^i_s)$, to compute two sets of fractions between 0 and 1 that each add up to 1. For instance, if three PMs provide packet processing capacity values of 1000, 1000, and 2000, respectively, then the normalized values for packet processing capacity are 0.25, 0.25 and 0.5. This achieves our "best effort" goal as we will invoke partitioning even when link bandwidths and end host CPU requirements cannot be supported on the available machines. We use the two sets of normalized values as METIS input parameters (partitioning constraints).

After graph partitioning, we compute ranking metrics and derive new inputs by potentially adding PMs (as explained in Section III-C4) and tuning the CPU shares (as explained in Section III-C5). The result of each iteration is stored in the hash set. A pseudo-code of the algorithm is given in Algorithm 1.

```
begin
    Initialize input queue;
    enqueue(init_input());
    while input queue non-empty do
        single_iteration (input queue.dequeue());
    end
    Output the best partition;
end
single_iteration (queue entry)
begin
    Compute packet processing capacities of PMs;
    host_in ← normalize (u^i_h);
    cap_in ← normalize (C^i(u^i_s));
    partition (G, host_in, cap_in);
    for PM i ∈ {1, ··· , k} do
        û^i_h ← Σ_{host j ∈ S_i.V} h_j;
        ĉ^i ← Σ_{v ∈ S_i.V} w(v), i.e., w(S_i);
        û^i_s ← smallest u such that C^i(u) ≥ ĉ^i;
    end
    evaluate_partition (û^i_h, û^i_s);
    u^i_s, u^i_h ← update_cpu_shares (û^i_h, û^i_s);
end
```

Algorithm 1: Waterfall Algorithm

3) Initial Input: The function *init_input* computes an initial input based on the given graph and the information of all available PMs. It consists of three phases: (1) composing a minimal set of chosen PMs, (2) allocating CPU shares for packet processing and end hosts for the chosen PMs, and (3) calculating parameters for METIS.

In the first phase, we compute the maximum resources each

available PM can offer, then compute and sort the resources in order of decreasing tightness. PMs are ranked by "usefulness", *i.e.*, at most how much a PM can contribute to each resource requirement (in tightness order). Tightness of a resource is defined as the ratio of the sum of the maximum amount of that resource each PM can offer to the needed amount of that resource. Next, we greedily grow the set of chosen PMs, which is initially empty, until all resource requirements are met or all PMs are included.

For example, consider an experiment that requires 1200 packet processing capacity shares and 100 CPU shares for emulated hosts. Consider a cluster with three PMs: PM1 can offer 1000 packet processing capacity or 200 CPU shares, PM2 offers 500 capacity or 200 CPU shares, and PM3 offers 200 capacity or 100 CPU shares. Then the first two PMs will be chosen since they provide a maximum of 1500 capacity or 400 CPU shares that satisfy both resource requirements. Note that this set of chosen PMs is a lower bound because (1) each resource requirement is considered independently whereas they affect each other, and (2) we are using the maximum possible offering from each PM to estimate resource availability.

In the second phase, the algorithm distributes CPU shares to packet processing and end hosts for each PM. Depending on the tightness of resources, it may assign most of CPU shares to packet processing and leave the rest to end hosts, or assign most CPU shares to support end hosts, or evenly allocate CPU shares to the two requirements. Users can modify this parameter in a configuration file.

In the third phase, the algorithm tweaks input parameters for METIS based on the result of previous phases. For instance, it searches, in parallel, for the best imbalance vector (a parameter that defines load imbalance tolerance for each constraint), first within a small range of $[1.0, 1.1] \times [1.0, 1.1]$, then a larger range of $[1.0, 1.5] \times [1.0, 1.5]$ if METIS shows non-trivial changes in edge cut. Finally, we initialize a termination counter for this input to a user-defined constant.

4) Evaluating Results: Based on the min cut value and assignment returned by METIS, we can compute the actual CPU shares and packet processing capacities of emulated hosts and switches assigned to PM i as follows: (1) $\hat{u}_h^i$: The actual CPU shares for emulated hosts assigned to this PM. (2) $\hat{c}^i$: The actual packet processing capacity for emulated switches assigned to this PM. (3) $\hat{u}_s^i$: The actual CPU shares needed to provide $\hat{c}^i$ packet processing capacity on this PM. (4) σ_h and σ_s: The fractions of host and packet processing capacities that this PM is assigned (over the total amount required). The total CPU usage of PM i for this assignment is then computed as $\hat{u} = \hat{u}_h^i + \hat{u}_s^i$.

An assignment is ranked according to the following factors: (1) Number of PMs used in the assignment, (2) Number of over-utilized PMs (*i.e.*, $\hat{u}^i > U^i$), (3) Number of under-utilized PMs (*i.e.*, $\hat{u}^i < U^i$), (4) Degree of over-utilization of PMs, as defined below in equation (3) when $\hat{u}^i > U^i$, and (5) The edge cut given by the graph partitioning, reflecting the total inter-PM traffic. The first four factors are derived from usage information of the assignment, and the fifth is directly returned by METIS. Typically, we use mutiplier thresholds for over- and under-utilization. These are user-defined constants, and are set to 100% and 90%, respectively, in our simulations and experiments.

$$over\text{–}util_i = \frac{\hat{u}^i - U^i}{U^i} \tag{3}$$

We rank each assignment according to three keys. The first key is a *tier*. If the assignment has no over-utilized PMs, it is considered tier 0; otherwise it is considered tier 1. For tier 0 assignments, we focus on reducing overall resource usage. Thus, the second key is the number of PMs used, and the third key is the edge cut of the assignment. For tier 1 assignments, we want to reduce the degree of over-utilization in order to reduce fidelity loss, so the second key is the maximum $over\text{–}util_i$, and the third key is the number of over-utilized PMs. The second key is used when there is a tie on the first key, and the third key is used when both the first and second keys are tied.

A multi-level hash set is implemented so that it takes constant time to query if an assignment has already appeared, whether an assignment is the best one, and if not, in which key is the assignment dominated by others.

When the majority of PMs are overloaded yet there is at least one unused PM, the algorithm will consider adding the next "most useful" PM, sorted as in Section III-C3, to the set of PMs chosen. The algorithm will construct a new input with initial shares and METIS parameters set in the same way as in phases 2 and 3 of Section III-C3. The termination counter for this new input is set to the initial value. We refer to this process as *branching* because it starts a new path for exploration. A threshold to determine "majority" (a user-defined constant) is used such that if the number of overloaded PMs is at least this fraction of the number of PMs chosen, branching will occur.

5) Updating CPU Shares: We update the termination counter based on the rank of the result, and if the updated counter is positive, we tweak the CPU share allocation and construct a new input. The rationale for decreasing the termination counter is discussed in Section III-C6.

To adjust CPU shares, we first sort the PMs by descending values of $(\sigma_h + \sigma_s)$. We then compute the CPU shares for the next iteration based on the output of the current iteration. The intuition is that, if a PM is overloaded, we assign it the maximum load it can handle and send the excessive shares to the next most powerful PM; if a PM is under-utilized, we increase its shares but no more than the shares added to any "stronger" under-utilized PM. The CPU share adjustment thus moves excessive CPU shares like a waterfall: overloaded shares flow towards the next most powerful PMs, and the room left for expansion in an under-utilized PM is limited. This is the reason we name this algorithm "Waterfall." The pseudo-code of the update algorithm is shown in Algorithm 2.

6) Algorithm Termination: The algorithm terminates when the input queue becomes empty, that is, when no new branches are created and all existing branches have exhausted their termination counter. The number of branches is upper-bounded by the number of PMs, and a branch stops after several iterations (a user-specified constant) without making progress. We use several heuristics when updating termination counter. For instance, if a new branch is created (*i.e.*, one more free PM is included) and all chosen PMs in the current input are over-utilized, then it is less likely to find the best assignment in this

begin
  total_over ← sum of excessive shares on all over-utilized PMs;
  Δ_{min} ← INT_MAX;
  for *PM i from highest to lowest ($\sigma_h + \sigma_s$)* **do**
    $\hat{u} \leftarrow \hat{u}^i_h + \hat{u}^i_s$;
    if *PM is over-utilized* **then**
      next_pp$_i$ ← min(max($\frac{U^i \times \hat{u}^i_s}{\hat{u}_i}$, U^i_{smin}), U^i_{smax});
      next_host$_i$ ← U^i − next_pp$_i$;
    else
      if *total_over > 0* **then**
        shares_over ← $\hat{u}_i$+ total_over $-U^i$;
        if *shares_over > 0* **then**
          Δ ← total_over − shares_over;
          total_over ← shares_over;
        else
          Δ ← total_over;
          total_over ← 0;
        end
      else
        $\Delta \leftarrow 0$;
      end
      if *PM under-utilized after adding Δ shares* **then**
        $\Delta \leftarrow$ min(Δ_{min}, max($\Delta, UnderThd \times U_i - \hat{u}_i$));
        $\Delta_{min} \leftarrow \Delta$;
      end
      $\Delta^i_h, \Delta^i_s \leftarrow \Delta \times \frac{\hat{u}^i_h}{\hat{u}^i_h + \hat{u}^i_s}, \Delta \times \frac{\hat{u}^i_s}{\hat{u}^i_h + \hat{u}^i_s}$;
      next_pp$_i$ ← min(max($\hat{u}^i_s + \Delta^i_s, U^i_{\text{smin}}$), U^i_{smax});
      next_host$_i$ ← min($\hat{u}^i_h + \Delta^i_h, U^i -$ next_pp$_i$);
    end
  end
  Return next_pp, next_host;
end

Algorithm 2: Waterfall Algorithm: CPU Share Update.

branch than in the new branch, so the termination counter is cut by half. If a new best assignment is found, the counter is reset; otherwise the counter is decreased by 4, 2, or 1, depending on how the rank compares to that of the best assignment.

Several factors affect the running time of the algorithm, but two factors play a major role: (1) tightness of available resources, and (2) characteristics of the experimental topology.

The tighter the resources, the smaller the search space. Tighter resources make our initial set of PMs closer to the set of PMs needed, resulting in fewer branches. Overloaded PMs are capped to their maximum CPU shares, and there is little tweaking the algorithm can do for them. In the case when all PMs have to be over-utilized, the algorithm will only run a few iterations.

The characteristics of the experimental topology are important when resources are abundant. For example, if a topology is highly clustered like a typical ISP topology, then, after increasing fractions of under-utilized PMs beyond certain levels, METIS will always group the clusters and assign an entire cluster to a single PM, making few changes to the assignment. Our algorithm will detect this situation and terminate early. In contrast, the algorithm will run substantially more iterations if the topology is more random and its vertices are "indistinguishable" in terms of resource requirements, because METIS is likely to swap vertices and return different – yet not better – assignments.

IV. Simulation Results

We have evaluated the Waterfall algorithm on a set of topologies, physical machine characteristics, and end host CPU requirements. In this section, we give simulation results on three different types of topologies ranging from 41 nodes to 690 nodes: RocketFuel, Jellyfish, and Fat-tree. To understand the impact of different PM characteristics, we evaluate all topologies in three scenarios: large clusters (resources are abundant), medium clusters (total maximum capacity of simulated PM cluster is close to the requirements (weight) of experimental topologies), and small clusters (total maximum capacity of simulated cluster is less than the requirements of experimental topologies). We scale up the four capacity functions in equation (1) in each cluster for large topologies.

We compare the Waterfall algorithm to four baseline algorithms: (1) Default METIS partitioning which assigns PMs approximately equal-weight partitions of the input graph; (2) Capacity-based partitioning, denoted by $C^i(0.9)$, assigns PM i a partition whose weight is proportional to $C^i(0.9 \times U^i)$ (*i.e.*, the capacity value when 90% of CPU shares are allocated to packet processing). The remaining 10% is reserved for end hosts and system overhead. This baseline is near optimal when packet processing needs most CPU shares; (3) Max CPU share-based partitioning, denoted by U^i, assigns PMs partitions proportional to the unscaled maximum CPU shares, U^i. For example, if two PMs have maximum CPU shares of 100 and 200, then the weights of the partitions are set to 0.33 and 0.67, respectively, regardless of their multiplier values; and (4) Scaled max CPU share-based partitioning, denoted by $\theta^i \times U^i$, is the same as U^i except that it uses scaled U^i values. If the two PMs in the previous example have multipliers of 2 and 1, respectively, then their scaled U^i values are equal, and thus they will get equally weighted partitions. We use METIS to compute partitions for all the baselines.

To compare the results of baselines with Waterfall, we use three metrics: (1) edge cut of the partition, (2) degree of over-utilization, defined by equation 3, and (3) degree of under-utilization, defined if $u^i < U^i$, as $under\text{–}util_i = \frac{U^i - u^i}{U^i}$.

The degree of over-utilization is the most important metric since overloaded PMs may result in fidelity loss. If there is under-utilization, it is possible that not all selected PMs are necessary for this experiment. The unnecessary PMs can be used by other experimenters in parallel. The edge-cut is an indicator of traffic localization as we discussed in section III-C. Ideally, we want to keep all three metrics as small as possible.

A. Large Clusters

In this scenario, we simulate a cluster with 21 PMs by duplicating capacity functions in equation (1) and scaling them up by a factor of 10.

Fig. 4(a) shows the average over-utilization and under-utilization for different topologies. Since the simulated PM cluster is sufficiently large for every topology, both $C^i(0.9)$ and Waterfall achieve less than 2% over-utilization while other baseline algorithms yield 4% to 14% over-utilization on different types of topologies. However, Waterfall selects fewer PMs and exhibits smaller under-utilization, achieving higher resource efficiency. In contrast, baseline algorithms use all PMs in the simulated cluster. Additionally, baseline algorithms show 1.3X to 6.6X edge-cuts compared to Waterfall, since they tend to use all PMs and hence create more partitions.

B. Medium Clusters

In this scenario, we create a PM cluster for each topology such that the cluster capacity is approximately equivalent to the resource requirements of the topology.

Fig. 4(b) shows higher over-utilization for all baseline algorithms. Waterfall again achieves less than 1% over-utilization on both Jellyfish and Fat-tree topologies, but slightly higher over-utilization (5%) on RocketFuel topologies. This is because RocketFuel topologies are less symmetric than Jellyfish and Fat-tree and include some nodes with much higher requirements leading to worse partitioning results when resources are limited. Baseline algorithms show smaller under-utilization compared to the large cluster scenario as we limit PM resources, but they still waste more than 20% of the resources on some PMs while overloading the rest. Baseline algorithms exhibit similar edge-cuts (0.8X to 1.2X) as Waterfall, since all algorithms create similar numbers of partitions and use METIS to minimize edge-cut.

C. Small Clusters

When the physical resources of a PM cluster are insufficient, Waterfall yields a best effort assignment in proportion to PM capabilities (*e.g.*, maximum capacity) leading to balanced resource utilizations for all PMs. In this scenario, instead of average over-utilization and under-utilization, we use standard error of PM utilization ($\frac{u^i}{U^i}$), for each topology to evaluate partitioning algorithms.

All algorithms overload all PMs as expected. Fig. 4(c) shows that Waterfall yields significantly smaller standard error compared to baseline algorithms, indicating more balanced PM utilizations for all topologies. Baseline algorithms exhibit slightly smaller edge-cuts (0.78X to 0.97X) than Waterfall, since they consider edge-cut minimization as the highest priority. In contrast, Waterfall considers over-utilization and the number of allocated PMs to be of higher priority than edge-cut minimization.

V. Experimental Evaluation

In this section, we evaluate our complete framework on experiments with a distributed denial of service (DDoS) attack scenario inspired by the recently presented Crossfire attack [31]. This type of experiment is popular on the DETER testbed [7] and represents a worst-case for mapping, since it pushes the emulated network to the limit. We use a cluster in our lab that includes six PMs of four types of hardware configurations. The traffic processing capacity functions of the four types of PMs were given in equation (1).

A. Mapping Algorithms

In addition to the algorithms used in our simulations in Section IV, we also compare the Waterfall algorithm to the Mininet `SwitchBinPlacer` (SwitchBin). This is the default "Placer" of the Mininet cluster mode. This mapping algorithm maps switches and controllers of an experiment into equally-sized bins based on the number of PMs. It also attempts to place hosts and switches to which they are connected on the same PM.

MaxiNet allows users to set *share* in the configuration file for each PM in a cluster. In our experiments, $C^i(0.9)$, U^i, and $\theta^i \times U^i$ are used when assigning shares to the PMs. Thus, `SwitchBinPlacer` and "Equal" are balanced partitioning algorithms which try to map either an equal number or equal weight of nodes to all PMs. $C^i(0.9)$, U^i, $\theta^i \times U^i$ and Waterfall are unbalanced partitioning algorithms that map nodes based on different functions of PM capacity. $C^i(0.9)$, U^i, and $\theta^i \times U^i$ utilize all PMs in a cluster, since they cannot determine if fewer PMs are sufficient for an experiment, while Waterfall picks the smallest number of PMs that are sufficient. Waterfall takes both the weights of switches (assigned based on link bandwidths) and the CPU shares of end hosts into consideration, and tries to balance the CPU usage among end hosts and software switches.

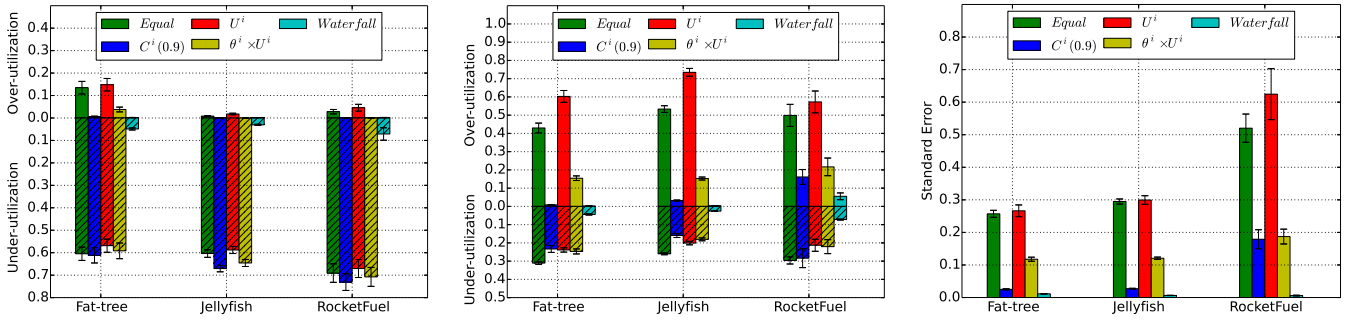
B. DDoS Attack Experiments

We design DDoS attack experiments inspired by the Crossfire attack [31] to stress-test the mapping algorithms using high rate traffic. Instead of attacking a web server directly, the attack targets critical links on the paths to the web server, and saturates these links to degrade the user experience to the victim web server. We compare three metrics: CPU utilizations of PMs, utilizations of all experimental topology links, and HTTP throughput (before and after the attack is launched). All experiments are repeated 10 times and error bars are shown.

1) Topology Generation: To make the DDoS attack experiment realistic but feasible to execute on our small testbed, we reduce ISP topologies from RocketFuel [32] to a set of small-scale topologies and medium-scale topologies, containing 10-15 routers and 30-50 routers, respectively. We use the random match (RM) algorithm for graph coarsening (used in [11]) to reduce the ISP topologies, while attempting to preserve the connectivity features of the original graph.

Each node in the reduced ISP topology is emulated as a switch. Link delays are set to 1 ms in the results below, but we also investigated 100 and 500 ms links. We attach end hosts to the "edges of the topology." The edge of a topology is defined as the nodes with degree ≤ 3, where degree is the number of edges incident onto a node. These end hosts are used as victim clients and attack senders. The same methodology was followed in [1].

2) Attack and Victim Host Assignment: There are four roles for nodes in a DDoS attack experiment: victim web server, victim clients, attack senders, attack receivers. We only assign one victim web server in each DDoS experiment. Victim clients are the hosts sending HTTP requests to the victim web server, and they are affected by DDoS attacks. Attack senders are the hosts launching DDoS traffic. As in the Crossfire



(a) Large cluster results (b) Medium cluster results (c) Small cluster results

Fig. 4. Simulation results

attack [31], we set the receivers of the attack traffic to be other servers close to the victim. These receivers are referred to as attack receivers in our experiment.

Prior to assigning hosts as attackers and victims, we use Dijkstra's algorithm to compute the shortest paths between any two hosts and use these paths as static routes (by default, Mininet does not include dynamic routing). We first rank all the nodes in a topology by node degree, and choose the node with median degree value to be where victim server is located, to avoid placing the victim server too close to the edge or to the center (*i.e.*, nodes with large degree) of a topology. The neighbors of the victim web server become attack receivers. We then compare routes from all edge hosts to the victim web server (victim clients to victim web server) with routes from all edge hosts to attack receiver candidates (attack senders to attack receivers) and assign hosts that share links in the two sets of routes as attack senders and victim clients. Then, $\sim$30% of the edge hosts are used as victim clients, and $\sim$70% are used as attack senders. Our goal here is to guarantee the effectiveness of attack traffic, *i.e.*, the web traffic will be affected when attack traffic is launched.

3) Traffic Generation: A DDoS experiment includes three types of traffic: HTTP traffic, UDP attack traffic, and background traffic. We use the *Web Polygraph* tool to generate HTTP traffic. Victim clients run *polygraph-client*, while the victim web server runs *polygraph-server*. *polygraph-client* generates HTTP requests to *polygraph-server* at a fixed rate (200 requests/sec per client). This rate may not be reached in an experiment if there is not enough bandwidth or CPU for *polygraph-client* processes. The HTTP response size generated by *polygraph-server* is exponentially distributed with a mean value of 10 KB. For attack traffic, we use *iperf* to generate UDP traffic at the same rate as the link bandwidth from attack senders to attack receivers. *iperf* is used to emulate background traffic. Both attack traffic and background traffic use a packet size of 1250 Bytes.

Both HTTP and background traffic start at the beginning of an experiment and we wait for 60 seconds for conditions to stabilize. Then, UDP attack traffic is launched and lasts for 60 seconds until the experiment is completed.

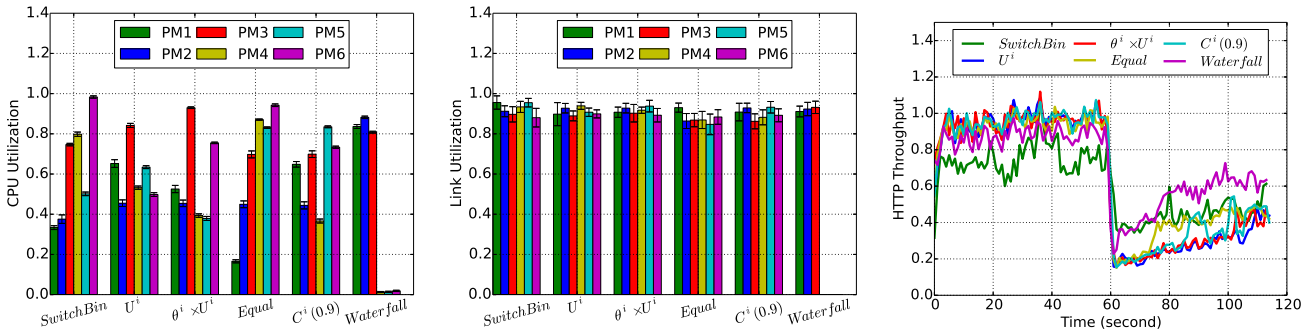
4) Small-scale Experiments: The motivation behind creating small-scale topologies is to compare Waterfall with other mapping algorithms when resources are over-provisioned. Since it is difficult to obtain the ground truth in network emulation, we use resource over-provisioning to approximate the ground truth results. For small topologies, baseline algorithms will use all 6 PMs in our cluster. Results are not affected by resource constraints. In contrast, Waterfall attempts to allocate the fewest PMs necessary. Therefore, if the results yielded by the 6 mapping algorithms are close, we consider Waterfall to have been able to produce close-to-ground-truth results while using resources more judiciously.

We carefully calculate the topology size, link bandwidths, and CPU shares of end hosts to guarantee that baseline algorithms do not overload any PM in this case. Based on this, link bandwidth is set to 60 Mbps. In this experiment, Waterfall takes only five iterations and chooses PM1 and PM2.

CPU and Link Utilization: Fig. 5(a) and Fig. 5(b) show the CPU and link utilizations. The baseline algorithms use all 6 PMs, and none of the PMs are overloaded. Waterfall picks the three most powerful PMs, and CPU utilizations of all PMs are over 80%. Therefore, we confirm that the results from baseline algorithms approximate the ground truth, since they are not constrained by PM resources. Note that the total CPU utilization over all six PMs for baseline algorithms seems significantly higher than Waterfall. This is caused by the heterogeneity of the six PMs. Running the same task on a less powerful PM usually consumes more CPU shares than on a more powerful PM. A key motivation of our capacity function is to capture this PM heterogeneity. All algorithms achieve at least 80% link utilization, indicating that sufficient resources are assigned to the switches. This is expected for baseline algorithms since resources are over-provisioned. For Waterfall, this confirms that our capacity functions accurately characterized the traffic processing capacity, and the mapping process allocated PMs appropriately, leading to the expected high experimental link utilization in this scenario, while maintaining a high CPU utilization level.

HTTP Throughput: HTTP throughput is the number of HTTP requests completed per second. We use this metric as an indicator of application-level performance fidelity. Fig. 5(c) shows the impact of the DDoS attack. At the 60th second mark, the attack traffic is launched, causing significant drop in HTTP throughput. The results of all mapping algorithms are similar, which indicates that our framework with the Waterfall algorithm is able to maintain high application-level fidelity.

5) Medium-scale Experiments: We design these experiments such that an ineffective mapping algorithm would suffer



(a) CPU utilization (b) Link utilization (c) Completed HTTP requests/second

Fig. 5. Normalized results from small DDoS attack experiments

from performance fidelity loss. We calculate the topology size, link bandwidths, and end host CPU shares such that the total requirements of the topology are slightly lower than the total capacity of the testbed. Link bandwidth is thus 16 Mbps in this case. If the mapping algorithm makes poor choices, PMs can be overloaded, and HTTP throughput may decrease. Ideally, CPU usage should be less than 100%, link utilizations should be high since we saturate the links, and HTTP throughput should exhibit a significant drop when DDoS attack traffic is launched.

CPU and Link Utilization: Fig. 6(a) and Fig. 6(b) show the CPU and link utilizations of the 6 mapping algorithms. SwitchBin over-utilizes PM5 and PM6 and under-utilizes PM1 and PM2. Only PM4 is assigned appropriate workload. PM3 exhibits low CPU and link utilization due to the fact that several flows from PM5 and PM6, which are overloaded, to PM3 experience significant packet loss (link fidelity loss). Therefore, PM4 does not process expected amounts of traffic leading to low CPU and link utilization. Both U^i and $\theta^i \times U^i$ over-utilize PM3 which has high CPU utilization but low link utilization with high variance. In addition, U^i over-utilizes PM6. Equal over-utilizes PM6 and under-utilizes PM1 and PM2 significantly, while $C^i(0.9)$ over-utilizes PM6 and under-utilizes PM4. Waterfall selects only five PMs, but offers the best CPU and link utilizations compared to baseline algorithms, with $> 80\%$ CPU utilization and $> 90\%$ link utilization on all five PMs. We also tried manually limiting baseline algorithms to use the same 5 PMs as Waterfall for a head-to-head comparison. However, none of them exhibited better performance than when they used all 6 PMs.

HTTP Throughput: Fig. 6(c) shows that the effect of the attack can be observed in all cases. Waterfall achieves higher and more stable HTTP throughput before the DDoS attack is launched. After careful examination, we found that this is due to the limited CPU resources given to the *polygraph-server* process in the case of the baseline algorithms. The required CPU share for each host is 10%. Since none of the baseline algorithms consider the end host CPU shares, most of the end hosts cannot receive enough CPU. For instance, the host running *polygraph-server* only receives 3% CPU on average when using $C^i(0.9)$. Even though $C^i(0.9)$ uses all 6 PMs, the actual HTTP throughput is still lower than Waterfall. Waterfall also does not achieve 100% target HTTP throughput: we found that this is due to the imperfect CPU isolation of Linux containers when the same CPU is shared by a number of different containers.

VI. Conclusions

In this paper, we have proposed a complete framework for efficiently mapping a networked application onto a cluster of (possibly heterogeneous) physical machines. Although we have focused on network experiments on the popular Mininet network emulator, in the future, we plan to extend our work to map other distributed applications onto distributed simulators, network testbeds, and data centers in general.

We have devised a resource quantification process to profile the physical machines, and designed and implemented a mapping algorithm, Waterfall, that takes both link bandwidths (via the edge/vertex weights in the input graph) and end host CPU requirements into consideration. The Waterfall algorithm attempts to use as few of the physical machines as possible, while achieving high performance fidelity. Based on results from simulations and DDoS attack testbed experiments, we find that our approach performs well in terms of both performance fidelity and testbed resource utilization. We are currently generalizing our framework to handle networks with multi-homed hosts and to support multiple resource limits. Finally, we are conducting extensive experiments, and making refinements to speed up the convergence of our algorithm, even in case of abundant resources and random topologies.

Acknowledgments

This work has been sponsored in part by NSF grant CNS-1319924.

References

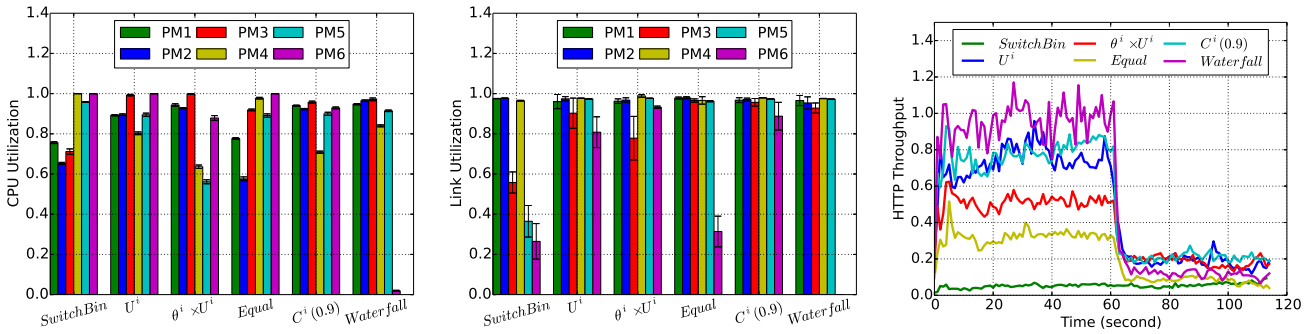
[1] W.-M. Yao, S. Fahmy, and J. Zhu, "EasyScale: Easy mapping for large-scale network security experiments," in *IEEE Conference on Communications and Network Security (CNS)*, Oct 2013, pp. 269–277.

[2] R. Chertov and S. Fahmy, "Forwarding devices: From measurements to simulations," *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, vol. 21, no. 2, pp. 1–23, 2011.

[3] N. Handigol, B. Heller, V. Jeyakumar, B. Lantz, and N. McKeown, "Reproducible network experiments using container-based emulation," in *Proc. of CoNEXT*, 2012, pp. 253–264.

[4] B. Lantz, B. Heller, and N. McKeown, "A network in a laptop: Rapid prototyping for software-defined networks," in *Proc. of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, ser. Hotnets-IX, 2010, pp. 19:1–19:6.

[5] "GENI," http://www.geni.net/.



(a) CPU utilization (b) Link utilization (c) Completed HTTP requests/second

Fig. 6. Normalized results from medium DDoS attack experiments

[6] "DETER," http://www.isi.edu/deter/.

[7] J. Mirkovic, H. Shi, and A. Hussain, "Reducing allocation errors in network testbeds," in *Proc. of the ACM Conference on Internet Measurement*, 2012, pp. 495–508.

[8] "Emulab," http://www.emulab.net/.

[9] P. Wette, M. Drxler, A. Schwabe, F. Wallaschek, M. H. Zahraee, and H. Karl, "MaxiNet: Distributed emulation of software-defined networks," in *Proc. of IFIP Networking*, June 2014.

[10] G. Karypis and V. Kumar, "Multilevel algorithms for multi-constraint graph partitioning," in *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*, ser. SC '98. Washington, DC, USA: IEEE Computer Society, 1998, pp. 1–13. [Online]. Available: http://dl.acm.org/citation.cfm?id=509058.509086

[11] ——, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM J. Sci. Comput.*, vol. 20, no. 1, pp. 359–392, Dec. 1998.

[12] D. Gupta, K. V. Vishwanath, M. McNett, A. Vahdat, K. Yocum, A. Snoeren, and G. M. Voelker, "DieCast: Testing distributed systems with an accurate scale model," *ACM Transactions on Computer Systems (TOCS)*, vol. 29, no. 2, pp. 1–48, 2011.

[13] E. Weingärtner, F. Schmidt, H. Vom Lehn, T. Heer, and K. Wehrle, "SliceTime: A platform for scalable and accurate network emulation," in *Proc. of NSDI*, 2011.

[14] B. W. Kernighan and S. Lin, "An efficient heuristic procedure for partitioning graphs," *Bell System Technical Journal*, vol. 49, no. 2, pp. 291–307, 1970.

[15] A. Pothen, H. D. Simon, and K.-P. Liou, "Partitioning sparse matrices with eigenvectors of graphs," *SIAM J. Matrix Anal. Appl.*, vol. 11, no. 3, pp. 430–452, May 1990.

[16] B. Hendrickson and R. Leland, "An improved spectral graph partitioning algorithm for mapping parallel computations," *SIAM J. Sci. Comput.*, vol. 16, no. 2, pp. 452–469, Mar. 1995.

[17] ——, "A multi-level algorithm for partitioning graphs," in *Supercomputing, 1995. Proceedings of the IEEE/ACM SC95 Conference*, 1995, pp. 28–28.

[18] S. T. Barnard and H. D. Simon, "A fast multilevel implementation of recursive spectral bisection for partitioning unstructured problems," in *PPSC*, 1993, pp. 711–718.

[19] J.-J. Kuo, H.-H. Yang, and M.-J. Tsai, "Optimal approximation algorithm of virtual machine placement for data latency minimization in cloud systems," in *Proc. of INFOCOM*, April 2014, pp. 1303–1311.

[20] X. Li, J. Wu, S. Tang, and S. Lu, "Let's stay together: Towards traffic aware virtual machine placement in data centers," in *INFOCOM, 2014 Proc. IEEE*, April 2014, pp. 1842–1850.

[21] J. Jiang, T. Lan, S. Ha, M. Chen, and M. Chiang, "Joint VM placement and routing for data center traffic engineering," in *Proc. of INFOCOM*, March 2012, pp. 2876–2880.

[22] M. Chowdhury, M. R. Rahman, and R. Boutaba, "ViNEYard: Virtual network embedding algorithms with coordinated node and link mapping," *IEEE/ACM Transactions on Networking*, vol. 20, no. 1, pp. 206–219, Feb 2012.

[23] M. Yu, Y. Yi, J. Rexford, and M. Chiang, "Rethinking virtual network embedding: Substrate support for path splitting and migration," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 17–29, Mar. 2008.

[24] R. Ricci, C. Alfeld, and J. Lepreau, "A solver for the network testbed mapping problem," *SIGCOMM Comput. Commun. Rev.*, vol. 33, no. 2, Apr. 2003.

[25] DETER Team, "Building apparatus for multi-resolution networking experiment using containers," DeterLab, Tech. Rep. ISI-TR-683, 2011.

[26] W.-M. Yao and S. Fahmy, "Flow-based partitioning of network testbed experiments," *Computer Networks*, vol. 58, pp. 141–157, January 2014.

[27] M. Hibler, R. Ricci, L. Stoller, J. Duerig, S. Guruprasad, T. Stack, K. Webb, and J. Lepreau, "Large-scale virtualization in the emulab network testbed," in *USENIX Annual Technical Conference*, 2008, pp. 113–128.

[28] J. Yan and D. Jin, "VT-Mininet: Virtual-time-enabled mininet for scalable and accurate software-define network emulation," in *Proc. of SOSR*, 2015, pp. 27:1–27:7.

[29] J. Pettit, J. Gross, B. Pfaff, M. Casado, and S. Crosby, "Virtual switching in an era of advanced edges," in *2nd Workshop on Data Center Converged and Virtual Ethernet Switching (DC-CAVES), ITC 22*, September 2010.

[30] G. Karypis and V. Kumar, "Metis – unstructured graph partitioning and sparse matrix ordering system, version 2.0," http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.38.376, Tech. Rep., 1995.

[31] M. S. Kang, S. B. Lee, and V. D. Gligor, "The crossfire attack," in *Proc. of the IEEE Symposium on Security and Privacy*, 2013, pp. 127–141.

[32] N. Spring, R. Mahajan, and D. Wetherall, "Measuring ISP topologies with rocketfuel," in *Proc. of SIGCOMM*, 2002, pp. 133–145.