# Scalable Rule Management for Data Centers

*Masoud Moshref*† *Minlan Yu*† *Abhishek Sharma*†∗ *Ramesh Govindan*†
† *University of Southern California* ∗ *NEC Labs America*

**Abstract**

Cloud operators increasingly need more and more fine-grained rules to better control individual network flows for various traffic management policies. In this paper, we explore automated rule management in the context of a system called vCRIB (a virtual Cloud Rule Information Base), which provides the abstraction of a centralized rule repository. The challenge in our approach is the design of algorithms that automatically off-load rule processing to overcome resource constraints on hypervisors and/or switches, while minimizing redirection traffic overhead and responding to system dynamics. vCRIB contains novel algorithms for finding feasible rule placements and adapting traffic overhead induced by rule placement in the face of traffic changes and VM migration. We demonstrate that vCRIB can find feasible rule placements with less than 10% traffic overhead even in cases where the traffic-optimal rule placement may be infeasible with respect to hypervisor CPU or memory constraints.

## 1 Introduction

To improve network utilization, application performance, fairness and cloud security among tenants in multi-tenant data centers, recent research has proposed many novel traffic management policies [8, 33, 29, 19]. These policies require *fine-grained* per-VM, per-VM-pair, or per-flow rules. Given the scale of today's data centers, the total number of rules within a data center can be hundreds of thousands or even millions (Section 2). Given the expected scale in the number of rules, rule processing in future data centers can hit CPU or memory resource constraints at servers (resulting in fewer resources for revenue-generating tenant applications) and rule memory constraints at the cheap, energy-hungry switches.

In this paper, we argue that future data centers will require *automated rule management* in order to ensure rule placement that respects resource constraints, minimizes traffic overhead, and automatically adapts to dynamics. We describe the design and implementation of a virtual Cloud Rule Information Base (vCRIB), which provides the *abstraction* of a centralized rule repository, and automatically manages rule placement without operator or tenant intervention (Figure 1). vCRIB manages rules for different policies in an integrated fashion even in the presence of system dynamics such as traffic changes or VM migration, and is able to manage a variety of data center configurations in which rule processing may be constrained either to switches or servers or may be permitted on both types of devices, and where both CPU and memory constraints may co-exist.

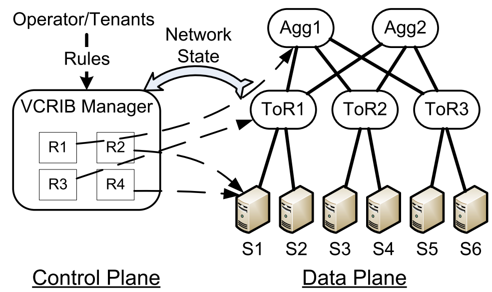



**Figure 1:** *Virtualized Cloud Rule Information Base (vCRIB)*

vCRIB's rule placement algorithms achieve resource-feasible, low-overhead rule placement by off-loading rule processing to nearby devices, thus trading off some traffic overhead to achieve resource feasibility. This trade-off is managed through a combination of three novel features (Section 3).

- Rule offloading is complicated by dependencies between rules caused by overlaps in the rule hyperspace. vCRIB uses per-source rule partitioning with replication, where the partitions encapsulate the dependencies, and replicating rules across partitions avoids rule inflation caused by splitting rules.
- vCRIB uses a *resource-aware placement* algorithm that offloads partitions to other devices in order to find a feasible placement of partitions, while also trying to co-locate partitions which share rules in order to optimize rule memory usage. This algorithm can deal with data center configurations in which some devices are constrained by memory and others by CPU.
- vCRIB also uses a *traffic-aware refinement algorithm* that can, either online, or in batch mode, refine partition placements to reduce traffic overhead while still preserving feasibility. This algorithm avoids local minima by defining novel benefit functions that perturb partitions allowing quicker convergence to feasi-

ble low overhead placement.

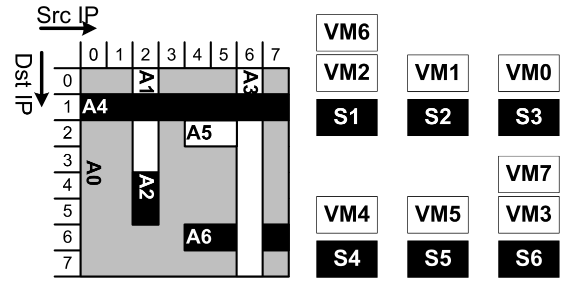
We evaluate (Section 4) vCRIB through large-scale simulations, as well as experiments on a prototype built on Open vSwitch [4] and POX [1]. Our results demonstrate that vCRIB is able to find feasible placements with a few percent traffic overhead, even for a particularly adversarial setting in which the current practice needs more memory than the memory capacity of all the servers combined. In this case, vCRIB is able to find a feasible placement, without relying on switch memory, albeit with about 20% traffic overhead; with modest amounts of switch memory, this overhead drops dramatically to less than 3%. Finally, vCRIB correctly handles heterogeneous resource constraints, imposes minimal additional traffic on core links, and converges within 5 seconds after VM migration or traffic changes.

## 2 Motivation and Challenges

Today, tenants in data centers operated by Amazon [5] or whose servers run software from VMware place their rules at the servers that source traffic. However, multiple tenants at a server may install too many rules at the same server causing unpredictable failures [2]. Rules consume resources at servers, which may otherwise be used for revenue-generating applications, while leaving many switch resources unused.

Motivated by this, we propose to automatically manage rules by offloading rule processing to other devices in the data center. The following paragraphs highlight the main design challenges in scalable automated rule management for data centers.

**The need for many fine-grained rules.** In this paper, we consider the class of data centers that provide computing as a service by allowing tenants to rent virtual machines (VMs). In this setting, tenants and data center operators need fine-grained control on VMs and flows to achieve different management *policies*. *Access control policies* either block unwanted traffic, or allocate resources to a group of traffic (e.g., rate limiting [33], fair sharing [30]). For example, to ensure each tenant gets a fair share of the bandwidth, Seawall [33] installs rules that match the source VM address and performs rate limiting on the corresponding flows. *Measurement policies* collect statistics of traffic at different places. For example, to enable customized routing for traffic engineering [8, 11] or energy efficiency [19], an operator may need to get traffic statistics using rules that match each flow (e.g., defined by five tuples) and count its number of bytes or packets. *Routing policies* customize the routing for some types of traffic. For example, Hedera [8] performs specific traffic engineering for large flows, while VLAN-based traffic management solutions [29] use different VLANs to route packets. Most of these policies,




(a) Wild card rules in a flow space (b) VM assignment

**Figure 2:** *Sample ruleset (black is accept, white is deny) and VM assignment (VM number is its IP)*

expressed in high level languages [20, 38], can be translated into virtual rules at switches[1].

A simple policy can result in a large number of fine-grained rules, especially when operators wish to control individual virtual machines and flows. For example, bandwidth allocation policies require one rule per VM pair [30] or per VM [30], and access control policies might require one rule per VM pair [31]. Data center traffic measurement studies have shown that 11% of server pairs in the same rack and 0.5% of inter-rack server pairs exchange traffic [24], so in a data center with 100$K$ servers and 20 VMs per server, there can, be 1$G$ to 20$G$ rules in total (200$K$ per server) for access control or fair bandwidth allocation. Furthermore, state-of-the-art solutions for traffic engineering in data centers [8, 11, 19] are most effective when *per-flow* statistics are available. In today's data centers, switches routinely handle between 1K to 10K active flows within a one-second interval [10]. Assume a rack with 20 servers and if each server is the source of 50 to 500 active flows, then, for a data center with 100$K$ servers, we can have up to 50$M$ active flows, and need one measurement rule per-flow.

In addition, in a data center where multiple concurrent policies might co-exist, rules may have dependencies between them, so may require carefully designed offloading. For example, a rate-limiting rule at a source VM A can overlap with the access control rule that blocks traffic to destination VM B, because the packets from A to B match both rules. These rules cannot be offloaded to different devices.

**Resource constraints.** In modern data centers, rules can be processed either at servers (hypervisors) or programmable network switches (e.g., OpenFlow switches). Our focus in this paper is on flow-based rules that match packets on one or more header fields (e.g., IP addresses, MAC addresses, ports, VLAN tags) and perform various actions on the matching packets (e.g., drop, rate limit, count). Figure 2(a) shows a flow-space with source and

[1] Translating high-level policies to fine-grained rules is beyond the scope of our work.

destination IP dimensions (in practice, the flow space has 5 dimensions or more covering other packet header fields). We show seven flow-based rules in the space; for example, $A1$ represents a rule that blocks traffic from source IP 2 (VM2) to destination IP 0-3 (VM 0-3).

While software-based hypervisors at servers can support complex rules and actions (e.g., dynamically calculating rates of each flow [33]), they may require committing an entire core or a substantial fraction of a core at each server in the data center. Operators would prefer to allocate as much CPU/memory as possible to client VMs to maximize their revenue; e.g., RackSpace operators prefer not to dedicate even a portion of a server core for rule processing [6]. Some hypervisors offload rule processing to the NIC, which can only handle limited number of rules due to memory constraints. As a result, the number of rules the hypervisor can support is limited by the available CPU/memory budget for rule processing at the server.

We evaluate the numbers of rules and wildcard entries that can be supported by Open vSwitch, for different values of flow arrival rates and CPU budgets in Figure 3. With 50% of a core dedicated for rule processing and a flow arrival rate of 1K flows per second, the hypervisor can only support about 2K rules when there are 600 wildcard entries. This limit can easily be reached for some of the policies described above, so that manual placement of rules at sources can result in *infeasible* rule placement.

To achieve feasible placement, it may be necessary to offload rules from source hypervisors to other devices and redirect traffic to these devices. For instance, suppose VM2, and VM6 are located on S1 (Figure 2(b)). If the hypervisor at S1 does not have enough resources to process the deny rule $A3$ in Figure 2(a), we can install the rule at ToR1, introducing more traffic overhead. Indeed, some commercial products already support offloading rule processing from hypervisors to ToRs [7]. Similarly, if we were to install a measurement rule that counts traffic between S1 and S2 at Aggr1, it would cause the traffic between S1 and S2 to traverse through Aggr1 and then back. The central challenge is to design a collection of algorithms that manages this tradeoff — keeps the traffic overhead induced by rule offloading low, while respecting the resource constraint.

Offloading these rules to programmable switches, which leverage custom silicon to provide more scalable rule processing than hypervisors, is also subject to resource constraints. Handling the rules using expensive power-hungry TCAMs limits the switch capacity to a few thousand rules [16], and even if this number increases in the future its power and silicon usage limits its applicability. For example, the HP ProCurve 5406zl switch hardware can support about 1500 OpenFlow wildcard rules using TCAMs, and up to 64K Ethernet forwarding entries [16].

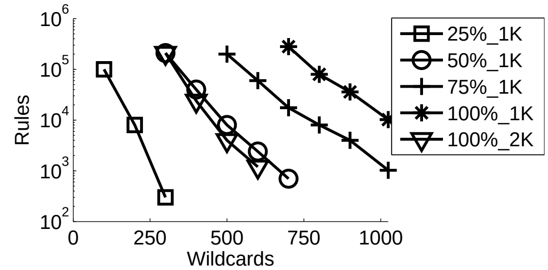



**Figure 3:** *Performance of openvswitch (The two numbers in the legend mean CPU usage of one core in percent and number of new flows per second.)*

**Heterogeneity and dynamics.** Rule management is further complicated by two other factors. Due to the different design tradeoffs between switches and hypervisors, in the future different data centers may choose to support either programmable switches, hypervisors, or even, especially in data centers with large rule bases, a combination of the two. Moreover, existing data centers may replace some existing devices with new models, resulting in device heterogeneity. Finding feasible placements with low traffic overhead in a large data center with different types of devices and qualitatively different constraints is a significant challenge. For example, in the topology of Figure 1, if rules were constrained by an operator to be only on servers, we would need to automatically determine whether to place a measurement rule for tenant traffic between S1 and S2 at one of those servers, but if the operator allowed rule placement at any device, we could choose between S1, ToR1, or S2; in either case, the tenant need not know the rule placement technology.

Today's data centers are highly dynamic environments with policy changes, VM migrations, and traffic changes. For example, if VM2 moves from $S1$ to $S3$, the rules $A0$, $A1$, $A2$ and $A4$ should me moved to $S3$ if there are enough resources at $S3$'s hypervisor. (This decision is complicated by the fact that $A4$ overlaps with $A3$.) When traffic changes, rules may need to be re-placed in order to satisfy resource constraints or reduce traffic overhead.

## 3 vCRIB Automated Rule Management

To address these challenges, we propose the design of a system called vCRIB (virtual Cloud Rule Information Base) (Figure 1). vCRIB provides the abstraction of a centralized repository of rules for the cloud. Tenants and operators simply install rules in this repository. Then vCRIB uses network state information including network topology and the traffic information to *proactively* place rules in hypervisors and/or switches in a way that respects resource constraints and minimizes the redirection traffic. Proactive rule placement incurs less controller overhead and lower data-path delays than a *purely reac-*

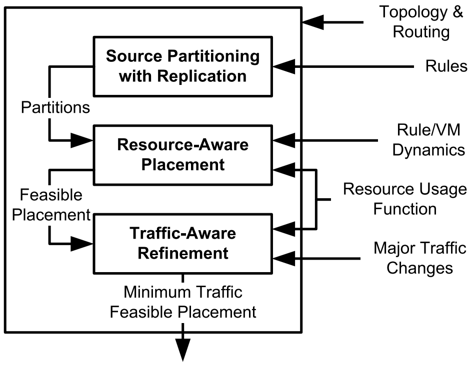



**Figure 4:** *vCRIB controller architecture*

*tive* approach, but needs sophisticated solutions to optimize placement and to quickly adapt to cloud dynamics (e.g., traffic changes and VM migrations), which is the subject of this paper. A hybrid approach, where some rules can be inserted reactively, is left to future work.

| Designs \ Challenges | Overlapping rules | Resource constraints | Traffic overhead | Heterogeneity | Dynamics |
|---|---|---|---|---|---|
| Partitioning with replication | ■ | ■ | | | |
| Per-source partitions | | ■ | ■ | | |
| Similarity | | ■ | | | |
| Resource usage functions | | | | ■ | |
| Resource-aware placement | | ■ | | | ■ |
| Traffic-aware refinement | | | ■ | | ■ |

**Table 1:** *Design choices and challenges mapping*

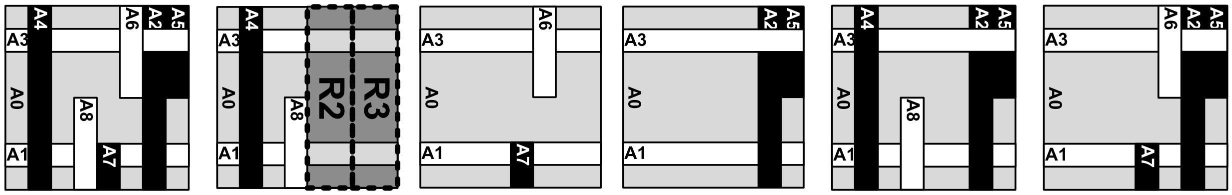
vCRIB makes several carefully chosen design decisions (Figure 4) that help address the diverse challenges discussed in Section 2 (Table 1). It partitions the rule space to break dependencies between rules, where each partition contains rules that can be co-located with each other; thus, a partition is the unit of offloading decisions. Rules that span multiple partitions are *replicated*, rather than split; this reduces rule inflation. vCRIB uses *per-source* partitions: within each partition, all rules have the same VM as the source so only a single rule is required to redirect traffic when that partition is offloaded. When there is *similarity* between co-located partitions (i.e., when partitions share rules), vCRIB is careful not to double resource usage (CPU/memory) for these rules, thereby scaling rule processing better. To accommodate device heterogeneity, vCRIB defines *resource usage functions* that deal with different constraints (CPU, memory etc.) in a uniform way. Finally, vCRIB splits the task of finding "good" partition off-loading opportunities into two steps: a novel bin-packing heuristic for *resource-aware partition placement* identifies feasible partition placements that respect resource constraints, and leverage similarity; and a fast *online traffic-aware refinement* algorithm which migrates partitions between devices to explore only feasible solutions while reducing traffic overhead. The split enables vCRIB to quickly adapt to small-scale dynamics (small traffic changes, or migration of a few VMs) without the need to recompute a feasible solution in some cases. These design decisions are discussed below in greater detail.

### 3.1 Rule Partitioning with Replication

The basic idea in vCRIB is to offload the rule processing from source hypervisors and allow more *flexible* and *efficient* placement of rules at both hypervisors and switches, while respecting resource constraints at devices and reducing the traffic overhead of offloading. Different types of rules may be best placed at different places. For instance, placing access control rules in the hypervisor (or at least at the ToR switches) can avoid injecting unwanted traffic into the network. In contrast, operations on the aggregates of traffic (e.g., measuring the traffic traversing the same link) can be easily performed at switches inside the network. Similarly, operations on inbound traffic from the Internet (e.g., load balancing) should be performed at the core/aggregate routers. Rate control is a task that can require cooperation between the hypervisors and the switches. Hypervisors can achieve end-to-end rate control by throttling individual flows or VMs [33], but in-network rate control can directly avoid buffer overflow at switches. Such flexibility can be used to manage resource constraints by moving rules to other devices.

However, rules cannot be moved unilaterally because there can be dependencies among them. Rules can overlap with each other especially when they are derived from different policies. For example, with respect to Figure 2, a flow from $VM6$ on server $S1$ to $VM1$ on server $S2$ matches both the rule $A3$ that blocks the source $VM1$ and the rule $A4$ that accepts traffic to destination $VM1$. When rules overlap, operators specify priorities so only the rule with the highest priority takes effect. For example, operators can set $A4$ to have higher priority. Overlapping rules make automated rule management more challenging because they constrain rule placement. For example, if we install $A3$ on S1 but $A4$ on ToR1, the traffic from $VM6$ to $VM1$, which should be accepted, matches $A3$ first and gets blocked.

One way to handle overlapping rules is to divide the flow space into multiple partitions and split the rule that intersects multiple partitions into multiple independent rules, *partition-with-splitting* [39]. Aggressive rule splitting can create many small partitions making it flexible to place the partitions at different switches [28], but can increase the number of rules, resulting in inflation. To minimize splitting, one can define a few large partitions, but these may reduce placement flexibility, since some partitions may not "fit" on some of the devices.




(a) Ruleset (b) Partition-with-replication (c) P1 & P3 on a device (d) P2 & P3 on a device

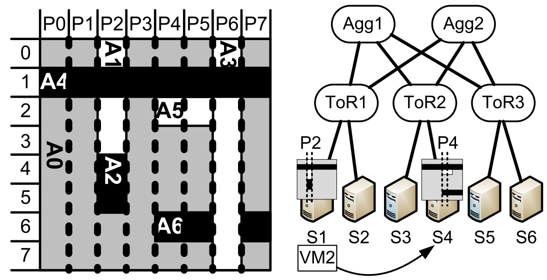
**Figure 5:** *Illustration of partition-with-replications (black is accept, white is deny)*

To achieve the flexibility of small partitions while limiting the effect of rule inflation, we propose a *partition-with-replication* approach that replicates the rules across multiple partitions instead of splitting them. Thus, in our approach, each partition contains the original rules that are covered partially or completely by that partition; these rules are not modified (e.g., by splitting). For example, considering the rule set in Figure 5(a), we can form the three partitions shown in Figure 5(b). We include both $A1$ and $A3$ in $P1$, the left one, in their original shape. The problem is that there are other rules (e.g., $A2$, $A7$) that overlap with $A1$ and $A3$, so if a packet matches $A1$ at the device where $P1$ is installed, it may take the wrong action – $A1$'s action instead of $A7$'s or $A2$'s action. To address this problem, we leverage redirection rules $R2$ or $R3$ at the source of the packet to completely cover the flow space of $P2$ or $P3$, respectively. In this way, any packets that are outside $P1$'s scope will match the redirection rules and get directed to the current host of the right partition where the packet can match the right rule. Notice that the other alternatives described above also require the same number of redirection rules, but we leverage high priority of the redirection rules to avoid incorrect matches.

Partition-with-replication allows vCRIB to flexibly manage partitions without rule inflation. For example, in Figure 5(c), we can place partitions $P1$ and $P3$ on one device; the same as in an approach that uses small partitions with rule splitting. The difference is that since $P1$ and $P3$ both have rules $A1$, $A3$ and $A0$, we only need to store 7 rules using partition-with-replication instead of 10 rules using small partitions. On the other hand, we can prove that the total number of rules using partition-with-replication is the same as placing one large partition per device with rule splitting (proof omitted for brevity).

vCRIB generates *per-source* partitions by cutting the flow space based on the source field according to the source IP addresses of each virtual machine. For example, Figure 6(a) presents eight per-source partitions $P0, \cdots, P7$ in the flow space separated by the dotted black lines.

Per-source partitions contain rules for traffic sourced by a single VM. Per-source partitions make the placement and refinement steps simpler. vCRIB only needs




(a) Per-source partitions (b) partition assignment

**Figure 6:** *Rule partition example*

one redirection rule installed at the source hypervisor to direct the traffic to the place where the partition is stored. Unlike per-source partitions, a partition that spans multiple source may need to be replicated; vCRIB does not need to replicate partitions. Partitions are ordered in the source dimension, making it easy to identify similar partitions to place on the same device.

### 3.2 Partition Assignment and Resource Usage

The central challenge in vCRIB design is the assignment of partitions to devices. In general, we can formulate this as an optimization problem, whose goal is to minimize the total traffic overhead subject to the resource constraints at each device.[2]

Suppose there are $P$ partitions where the $i$th partition has $|P_i|$ rules, and $N$ devices each with the memory size $S_j$ ($j = 1..N$). An indicator variable $M_{i,j}$ denotes that partition $P_i$ is placed on device $j$. To calculate the traffic overhead for placing partition $P_i$ on device $j$, for each flow $f$ matching $P_i$, we calculate the traffic ($T$) of directing $f$ to $j$ versus placing the partition at $s(f)$ (the source of $f$): $T^f_{i,j} - T^f_{i,s(f)}$.

The goal of the optimization is to minimize the total traffic overhead $\sum_{i=1..P, j=1..N} \sum_{f \text{ match } P_i} M_{i,j}(T^f_{i,j} - T^f_{i,s(f)})$. The resource constraint is that the total number of rules in the partitions at any device should be less than the device memory capacity. (in this formulation, we consider only memory constraints, and later generalize this to CPU constraints). For the approaches

[2] One may formulate other optimization problems such as minimizing the resource usage given the traffic usage budget. A similar greedy heuristic can also be devised for these settings.

that use small partitions with splitting, or one large partition, this is $\sum_i M_{i,j} * |P_i| \leq S_j$ while for our partition-with-replication approach, this constraint is $\left|\bigcup_{i,M_{i,j}=1} P_i\right| \leq S_j$. This problem, even for *partition-with-splitting*, is equivalent to the *generalized assignment problem*, which is NP-hard and even APX-hard to approximate [14]. Although there are some pseudo-polynomial approximation algorithms, they are inefficient for objects with similarity. For example, the algorithm in [15] performs the Knapsack dynamic programming (DP) algorithm for each device. The complexity of this approach is large for our problem not only because of its pseudo-polynomial complexity but also because of similarity between objects: In each step, we need to check all solutions in the previous row of DP table as we do not know adding the new partition to which previous solution will be feasible.

We propose a two-step heuristic algorithm to solve this problem. First, we perform *resource-aware placement* of partitions, a step which only considers resource constraints; next, we perform *traffic-aware refinement*, a step in which partitions reassigned from one device to another to reduce traffic overhead. An alternative approach might have mapped partitions to devices first to minimize traffic overhead (e.g., placing all the partitions at the source), and then refined the assignments to fit resource constraints. With this approach, however, we cannot guarantee that we can find a feasible solution in the second stage. Similar two-step approaches have also been used in the resource-aware placement of VMs across servers [22]. However, placing partitions is more difficult than placing VMs because it is important to co-locate partitions which share rules, and placing partitions at different devices incurs different resource usage.

Before discussing these algorithms, we describe how vCRIB models resource usage in hypervisors and switches in a uniform way. As discussed in Section 2, CPU and memory constraints at hypervisors and switches can impact rule placement decisions. We model resource constraints using a function $\mathscr{F}(P,d)$; specifically, $\mathscr{F}(P,d)$ is the percentage of the resource consumed by placing partition $P$ on a device $d$. $\mathscr{F}$ determines how many rules a device can store, based on the rule patterns (i.e., exact match, prefix-based matching, and match based on wildcard ranges) and the resource constraints (i.e., CPU, memory). For example, for a *hardware OpenFlow switch d* with $s_{TCAM}(d)$ TCAM entries and $s_{SRAM}(d)$ SRAM entries, the resource consumption $\mathscr{F}(P,d) = r_e(P)/s_{SRAM}(d) + r_w(P)/s_{TCAM}(d)$, where $r_e$ and $r_w$ are the numbers of exact matching rules and wildcard rules in $P$ respectively.

The resource function for *Open vSwitch* is more complicated and depends upon the number of rules $r(P)$ in the partition $P$, the number of wildcard patterns $w(P)$ in $P$, and the rate $k(d)$ of new flow arriving at switch $d$. Figure 3 shows the number of rules an Open vSwitch can support for different number of wild card patterns.[3] The number of rules it can support reduces exponentially with the increase of the number of wild card patterns (the y-axis in Figure 3 is in log-scale), because Open vSwitch creates a hash table for each wild card pattern and goes through these tables linearly. For a fixed number of wild card patterns and the number of rules, to double the number of new flows that Open vSwitch can support, we must double the CPU allocation.

We capture the CPU resource demand of Open vSwitch as a function of the number of new flows per second matching the rules in partition and the number of rules and wild card patterns handled by it. Using non-linear least squares regression, we achieved a good fit for Open vSwitch performance in Figure 3 with the function $\mathscr{F}(P,d) = \alpha(d) \times k(d) \times w(P) \times \log\left(\frac{\beta(d)r(P)}{w(P)}\right)$, where $\alpha = 1.3 \times 10^{-5}$, $\beta = 232$, with $R^2 = 0.95$.[4]

### 3.3 Resource-aware Placement

Resource-aware partition placement where partitions do not have rules in common can be formulated as a bin-packing problem that minimizes the total number of devices to fit all the partitions. This bin-packing problem is NP-hard, but there exist approximation algorithms for it [23]. However, resource-aware partition placement for vCRIB is more challenging since partitions may have rules in common and it is important to co-locate partitions with shared rules in order to save resources.

**Algorithm 1** First Fit Decreasing Similarity Algorithm

```
𝒫= set of not placed partitions
while |𝒫| > 0 do
    Select a partition P_i randomly
    Place P_i on an empty device M_k.
    repeat
        Select P_j ∈ 𝒫 with maximum similarity to P_i
    until Placing P_j on M_k Fails
end while
```

We use a heuristic algorithm for bin-packing similar partitions called *First Fit Decreasing Similarity* (FFDS) (Algorithm 1) which extends the traditional FFD algorithm [34] for bin packing to consider *similarity* between partitions. One way to define similarity between two partitions is as the number of rules they share. For example, the similarity between $P4$ and $P5$ is $|P4 \cap P5| =$

[3] The IP prefixes with different lengths 10.2.0.0/24 and 10.2.0.0/16 are two wildcard patterns. The number of wildcard patterns can be large when the rules are defined on multiple tuples. For example, the source and destination pairs can have at most 33*33 wildcard patterns.

[4] $R^2$ is a measure of *goodness of fit* with a value of 1 denoting a perfect fit.

$|P4| + |P5| - |P4 \cup P5| = 4$. However, different devices may have different resource constraints (one may be constrained by CPU, and another by memory). A more general definition of similarity between partitions $P_i$ and $P_k$ on device $d$ is based on the resource consumption function $\mathscr{F}$: our similarity function $\mathscr{F}(P_i, d) + \mathscr{F}(P_k, d) - \mathscr{F}(P_i \cup P_k, d)$ compares the network resource usage of co-locating those partitions.

Given this similarity definition, FFDS first picks a partition $P_i$ randomly and stores it in a new device.[5] Next, we pick partitions similar to $P_i$ until the device cannot fit more. Finally, we repeat the first step till we go through all the partitions.

When we are dealing with per-source partitions in the memory model case, we do not need to compute the similarity of partitions to find the most similar partition to another: Suppose that we have a sorted list of per-source partitions based on the source IP in ascending order and $P_i$ is the $i$th partition in this list. As rules are continuous boxes in the flow space if a rule is in partition $P_i$ and $P_k$ it must be present in $P_j$ if $i < j < k$.[6] So according to the definition of similarity of two partitions ($|P_1 \cap P_2|$), $P_j$ and $P_i$ similarity is larger than $P_k$ and $P_i$. Applying the conclusion for $k$ on the other way shows that $P_j$ is more similar to $P_k$ comparing to $P_i$:

$$|P_i \cap P_j| \geq |P_i \cap P_k| \quad if\ |i - j| \leq |i - k| \tag{1}$$

So closer partitions to $P_i$ are always more similar and the most similar ones are the neighbors in the sorted list: $P_{i-1}$ and $P_{i+1}$. Using this property, we can improve the duration of resource-aware placement step from 2 minutes to a seconds. For example, $P4$ has 4 common rules with $P5$ but 3 common rules with $P7$ in Figure 6(a). Therefore, when we place $P4$ on an empty device, we only need to check neighbor partitions $P5$ and $P3$.

To illustrate the algorithm, suppose each server in the topology of Figure 1 has a capacity of four rules to place the partitions and switches have no capacity. Considering the ruleset in Figure 2(a), we first pick a random partition $P4$ and place it on an empty device. Then, we check $P3$ and $P5$ and pick $P5$ as it has more similar rules (4 vs 2). Between $P3$ and $P6$, $P6$ is the most similar but the device has no additional capacity for $A3$, so we stop. In the next round, we place $P2$ on an empty device and bring $P1$, $P0$ and $P3$ but stop at $P6$ again. The last device will contain $P6$ and $P7$.

Our FFDS algorithm is inspired by the tree-based placement algorithm proposed in [34], which minimizes the number of servers to place VMs by putting VMs with more common memory pages together. There are three key differences: (1) since we use per-source partitions, it is easier to find the most similar partitions than memory pages; (2) instead of placing sub-trees of VMs in the same device, we place a set of similar partitions in the same device. These similar partitions are not bounded by the boundaries of a sub-tree; and (3) we are able to achieve a tighter approximation bound (2, instead of 3).

To prove the approximation bound, we first go through the procedure presented in [34] for creating a tree to capture similarity of partitions and assigning sub-trees to machines. Then we prove the binary tree guarantees 2-approximation of the *Fractional Packing* lower bound instead of 3-approximation for k-d tree. Fractional Packing approach relaxes the integrality of partitions and assumes that we can break them to arbitrary small pieces. However, the Fractional Packing solution can be worse than the optimal integral partition assignment if the tree cannot capture 100% similarities of partitions. In the second step, we prove that for the rules with prefix wildcards, the created tree captures 100% of similarity of partitions. Thus the Fractional Packing solution is the lower bound for optimal solution of placing integral partitions. At last, we prove that FFDS algorithm is as good as the binary tree in theory and show that it is better in many cases.

**Create the tree** Suppose that we have devices with rule capacity $M$ and we want to create a tree of nodes where each node $v$ is associated with a set of rules $R_v$, a capacity $cap_v$, an exclusive size $w_v$, a size $size_v$ and a count $c_v$.

Creating the tree has three phases: creating the tree-structure bottom-up, updating the capacities of nodes top-down, and updating the count values for nodes bottom-up. Capacities, sizes and counts are 0 by default. To create the tree we put partitions as the leaves of the tree. Then we create a tree structure out of them recursively by picking the most similar nodes, $v_i$ and $v_j$, and put them under a new node as the parent, $v_p$, with $R_{v_p} = R_{v_i} \cap R_{v_j}$ and set $w_{v_i} = |R_{v_i} - R_{v_p}|$ and $w_{v_j} = |R_{v_j} - R_{v_p}|$. In the second phase, we set the capacity of the root node to $M$. Going top-down on the tree from $v_p$ to its child node $v_i$, we set $cap_{v_i} = cap_{v_p} - w_{v_p}$. In the last phase, going up in the tree to the parent $v_p$ of $v_i$ and $v_j$, we set $c_{v_p} = \frac{size_{v_i} + size_{v_j}}{cap_{v_p} - w_{v_p}}$ and $size_{v_p} = size_{v_i} + size_{v_j} + c_{v_p} \times w_{v_p}$. After this procedure, $w_v$ will show the number of exclusive rules saved because of partitions under $v$ and are common among the partitions, $cap_v$ will show the capacity remained on a machine for rules saved exclusively because of partitions under $v$, $size_v$ will show the to-

[5] As a greedy algorithm, one would expect to pick large partitions first. However, since we have different resource functions for different devices, it is hard to pick the large partitions based on different metrics. Fortunately, in theory, picking partitions randomly or greedily do not affect the approximation bound of the algorithm. As an optimization, instead of picking a new device, we can pick the device whose existing rules are most similar to the new partition.

[6] Note that although the part of the rule in $P_j$ may be covered completely by other rules, in partitioning with replication method we do not remove it.

tal rule spaces needed to save those exclusive rules and most importantly, $c_v$ shows the number of devices required for saving partitions under node $v$ in *Fractional Packing* which is: 1) put each leaf node $v$ on a device with capacity $cap_v$ 2) Recursively, consolidate the devices of children $(v_i, v_j)$ of node $v_p$ into devices with capacity $cap_{v_p} - w_{v_p}$ and add $w_{v_p}$ exclusive rules to them. After each consolidation all $c_{v_p}$ devices (may be except one) will be full. The consolidation relies on the fact that we can break partitions to arbitrary sizes.

**Placement:** To place the partitions on the tree, we follow Algorithm 2 which is the revised version of the Greedy algorithm presented in [34] for the binary tree. In each round we find partitions under two most similar sub-trees that cannot be placed on a device and place them on two devices.

**Algorithm 2** Greedy Packing on tree

**while** tree has a node with count $> 1$ **do**
  Find a node, $v$ with count $c_v = 2$ where count of its children is 1.
  Place partitions in sub-trees of each children on one device
  Remove $v$ from tree and update *size* and $c$ of its ancestors
**end while**
Place partitions under root node on one device.

**2-approximation bound:** The Greedy Packing algorithm (G) uses at most two times devices used by Fractional Packing algorithm (F).

*Proof.* We use induction on the structure of tree after each round to prove the bound

$$G(T) \leq 2F(T) \tag{2}$$

The base case is correct as if $c_{root} = 1$, we use only one device. Suppose $T'$ is created by removing $v$ from the tree $T$ in the $n$th round of Algorithm 2. We add 2 devices each round so

$$G(T) = G(T') + 2 \tag{3}$$

Also the Fractional packing algorithm creates at least one full device for $v$ which will be untouched until the end of the algorithm.

$$F(T') \leq F(T) - 1 \tag{4}$$

Putting Equations 2, 3 and 4 together we get:

$$G(T) = G(T') + 2 \leq 2F(T') + 2 \leq 2F(T)$$

□

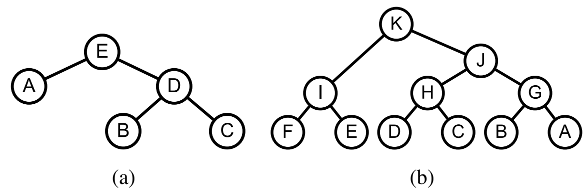



**Figure 7:** *The tree for Greedy Packing algorithm. (a) Tree captures 100% similarity (b) A Greedy Packing tree example*

**Fractional Packing is a true lower bound:** We just proved that we are in 2-approximation bound of the Fractional Packing method. However, the Fractional Packing solution can be larger than optimum if the tree cannot capture the 100% of similarity between partitions. For example, in Figure 7(a) nodes $B$ and $C$ where the most similar. The tree loses similarity of nodes in case $A$ has some similar rules with only $B$ or $C$. This similarity is not captured by node $D$ and then the node $E$. So the count value of $E$ can be larger than the optimal solution. However, we prove that for per-source partitions and prefix wildcard rules this situation does not happen. Therefore, Fractional Packing algorithm is the lower bound for the optimal solution. As we use per-source partitions, closer partitions are always the most similar ones (Equation 1), so each node has one or more partitions continuously picked from source IP dimension. Note that partitions in $A$ cannot be between $B$ and $C$ in source IP dimension as we know $B$ and $C$ are the most similar and similar nodes are the most close ones. Without loss of generality, assume partitions of node $C$ have larger source IP than of node $B$. Assume that there is a common rule, $R_1$, between $A$ and $B$. If partitions in $A$ have larger source IP than $C$, node $C$ must also have the rule $R_1$ as rules are continuous boxes. On the other hand, if partitions in $A$ have smaller source IP than $B$, it violates the prefix wildcards condition: As $B$ and $C$ are the most similar nodes, they must have at least one common rule, $R_2$. So $R_1$ and $R_2$ share a part of source IP dimension while neither covers another and this cannot happen if rules follow prefix wildcards on the source IP dimension.

**FFDS vs. Greedy Packing:** In the last step, we show that FFDS algorithm is as good as the Greedy Packing algorithm on the tree and it even works better in many cases. Consider a similarity tree, that the Greedy Packing algorithm put one of its sub-trees on a device. If FFDS starts from one of the partitions in that sub-tree, it will first go through the partitions in that sub-tree because they are most similar. Then, it may add some partitions from adjacent sub-trees. So it deals with an equal or smaller tree in every round. For example, suppose Greedy Packing created the similarity tree in Figure 2, for partitions with size 60 where each pair ($AB$, $CD$, $EF$)

have 10 common rules. The goal is to place partitions on devices with 170 rule capacities. Greedy Packing algorithm will select node $J$ as its count is two but the count of its children are one. So partitions $A$ and $B$ will be on one device and $C$ and $D$ on another. In the last step it will place $F$ and $E$ on the third device. However, FFDS may start from the partition $B$[7]. Next, it will put $A$ as it is the most similar to $B$. FFDS is not bounded by the sub-trees boundaries, so it checks partition $C$ in the next step. Putting partition $C$ on the first device will be successful as it still has 60 free spaces. Following the same approach in the next round, FFDS places partitions $D$, $E$ and $F$ on the second device. So FFDS used two devices while Greedy Packing needs three devices.

### 3.4 Traffic-aware Refinement

The resource-aware placement places partitions without heed to traffic overhead since a partition may be placed in a device other than the source, but the resulting assignment is *feasible* in the sense that it respects resource constraints. We now describe an algorithm that *refines* this initial placement to reduce traffic overhead, while still maintaining feasibility. Having thus separated placement and refinement, we can run the (usually) fast refinement after small-scale dynamics (some kinds of traffic changes, VM migration, or rule changes) that do not violate resource feasibility. Because each per-source partition matches traffic from exactly one source, the refinement algorithm only stores each partition *once* in the entire network but tries to migrate it closer to its source.

Given per-source partitions, an *overhead-greedy* heuristic would repeatedly pick the partition with the largest traffic overhead, and place it on the device which has enough resources to store the partition and the lowest traffic overhead. However, this algorithm cannot handle dynamics, such as traffic changes or VM migration. This is because in the steady state many partitions are already in their best locations, making it hard to rearrange other partitions to reduce their traffic overhead. For example, in Figure 6(a), assume the traffic for each rule (excluding A0) is proportional to the area it covers and generated from servers in topology of Figure 6(b). Suppose each server has a capacity of 5 rules and we put $P4$ on $S4$ which is the source of $VM4$, so it imposes no traffic overhead. Now if $VM2$ migrates from $S1$ to $S4$, we cannot save both $P2$ and $P4$ on $S4$ as it will need space for 6 rules, so one of them must reside on ToR2. As $P2$ has 3 units deny traffic overhead on $A1$ plus 2 units of accept traffic overhead from local flows of $S4$, we need to bring $P4$ out of its sweet spot and put $P2$ instead. However, the overhead-greedy algorithm cannot move $P4$ as it is already in its best location.

To get around this problem, it is important to choose a potential refinement step that not only considers the benefit of moving the selected partition, but also considers the other partitions that might take its place in future refinement steps. We do this by calculating the *benefit* of moving a partition $P_i$ from its current device $d(P_i)$ to a new device $j$, $M(P_i, j)$. The benefit comes from two parts: (1) The reduction in traffic (the first term of Equation 5); (2) The potential benefit of moving other partitions to $d(P_i)$ using the freed resources from $P_i$, excluding the lost benefit of moving these partitions to $j$ because $P_i$ takes the resources at $j$ (the second term of Equation 5). We define the potential benefit of moving other partitions to a device $j$ as the maximum benefits of moving a partition $P_k$ from a device $d$ to $j$, i.e., $Q_j = max_{k,d}(T(P_k, d) - T(P_k, j))$. We speed up the calculation of $Q_j$ by only considering the current device of $P_k$ and the best device $b(P_k)$ for $P_k$ with the least traffic overhead. (We omit the reasons for brevity.) In summary, the benefit function is defined as:

$$M(P_i, j) = (T(P_i, d(P_i)) - T(P_i, j)) + (Q_{d(P_i)} - Q_j) \quad (5)$$

**Algorithm 3** Benefit-Greedy algorithm

Update $b(P_i)$ and $Q(d)$
**while** not timeout **do**
  Update the benefit of moving every $P_i$ to its best feasible target device $M(P_i, b(P_i))$
  Select $P_i$ with the largest benefit $M(P_i, b(P_i))$
  Select the target device $j$ for $P_i$ that maximizes the benefit $M(P_i, j)$
  Update best feasible target devices for partitions and $Q$'s
**end while**
return the best solution found

Our traffic-aware refinement algorithm is *benefit-greedy*, as described in Algorithm 3. The algorithm is given a time budget (a "timeout") to run; in practice, we have found time budgets of a few seconds to be sufficient to generate low traffic-overhead refinements. At each step, it first picks that partition $P_i$ that would benefit the most by moving to its best feasible device $b(P_i)$, and then picks the most beneficial and feasible device $j$ to move $P_i$ to.[8]

We now illustrate the benefit-greedy algorithm (Algorithm 3) using our running example in Figure 6(b). The best feasible target device for both $P2$ and $P4$ are $ToR2$. $P2$ maximizes $Q_{S4}$ with value 5 because its deny traffic is 3 and has 1 unit of accept traffic to $VM4$ on $S4$. Also we assume that $Q_j$ is zero for all other devices. In the first step, the benefit of migrating $P2$ to $ToR2$ is larger than moving $P4$ to $ToR2$, while the benefits of all the other

[7] Starting from other partitions result the same conclusion.

[8] By feasible device, we mean the device has enough resources to store the partition according to the function $\mathscr{F}$.

migration steps are negative. After moving $P2$ to $ToR2$ the only beneficial step is moving $P4$ out of $S4$. After moving $P4$ to $ToR2$, migrating $P2$ to $S4$ become feasible, so $Q_{S4}$ will become 0 and as a result the benefit of this migration step will be 5. So the last step is moving $P2$ to $S4$.

A couple of optimizations to the benefit-greedy algorithm are possible. The algorithm may move a partition out of its "sweet spot" location, but it is not guaranteed to move another partition into that slot. To restore the best position for the partitions, we run the overhead-greedy algorithm at the end of each run of benefit-greedy while respecting the timeout. Instead of predicting if saving a partition will be feasible after bringing out another one, we change the Equation 5 to Equation 6 by estimating the average number of partitions, $\alpha$, we need to bring out from their sweet spot to save another. Large $\alpha$ signals that too many partitions must sacrifice for an infeasible partition to move to a better place. So Equation 6 will damp the effect of infeasible migrations.

$$M(P_i, j) = (T(P_i, d(P_i)) - T(P_i, j)) + \frac{1}{\alpha}(Q_{d(P_i)} - Q_j) \tag{6}$$

An alternative to using a greedy approach would have been to devise a randomized algorithm for perturbing partitions. For example, a Markov approximation method is used in [22] for VM placement. In this approach, checking feasibility of a partition movement to create the links in the Markov chain turns out to be computationally expensive. Moreover, a randomized iterative refinement takes much longer to converge after a traffic change or a VM migration.

# 4 Evaluation

We first use simulations on a large fat-tree topology with many fine-grained rules to study vCRIB's ability to minimize traffic overhead given resource constraints. Next, we explore how the online benefit-greedy algorithm handles rule re-placement as a result of VM migrations. Our simulations are run on a machine with quad-core 3.4 GHz CPU and 16 GB Memory. Finally, we deploy our prototype in a small testbed to understand the overhead at the controller, and end-to-end delay between detecting traffic changes and re-installing the rules.

## 4.1 Simulation Setup

**Topology:** Our simulations use a three-level fat-tree topology with degree 16, containing 1024 servers in 128 racks connected by 320 switches. Since current hypervisor implementations can support multiple concurrent VMs [32], we use 20 VMs per machine. We consider two models of resource constraints at the servers: memory constraints (e.g., when rules are offloaded to a NIC), and CPU constraints (e.g., in Open vSwitch). For switches, we only consider memory constraints.

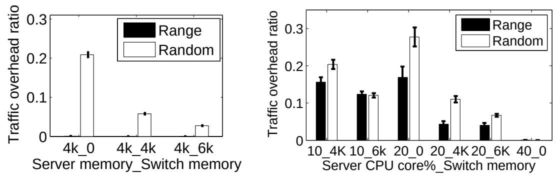
**Rules:** Since we do not have access to realistic data center rule bases, we use ClassBench [36] to create 200K synthetic rules each having 5 fields. ClassBench has been shown to generates rules representative of real-world access control.

**VM IP address assignment:** The IP address assigned to a VM determines the number of rules the VM matches. A random address assignment that is oblivious to the rules generated in the previous set may cause most of the traffic to match the default rule. The problem of selecting VM IPs such that the maximum number of rules are covered is equal to finding the Densest subhypergraph which is NP-Hard and hard to approximate [18]. Instead, we use a heuristic – we first segment the IP range with the boundaries of rules on the source and destination IP dimensions and pick random IP addresses from randomly chosen ranges. We test two arrangements: *Random* allocation which assigns these IPs randomly to servers and *Range* allocation which assigns a block of IPs to each server so the IP addresses of VMs on a server are in the same range.

**Flow generation:** Following prior work, we use a staggered traffic distribution (ToRP=0.5, PodP=0.3, CoreP=0.2) [8]. We assume that each machine has an average of 1K flows that are uniformly distributed among hosted VMs; this represents larger traffic than has been reported [10], and allows us to stress vCRIB. For each server, we select the source IP of a flow randomly from the VMs hosted on that machine and select the destination IP from one of the target machines matching the traffic distribution specified above. The protocol and port fields of flows also affect the distribution of used rules. The source port is wildcarded for ClassBench rules so we pick that randomly. We pick the destination port based on the protocol fields and the port distributions for different protocols (This helps us cover more rules and do not dwell on different port values for ICMP protocol.). Flow sizes are selected from a Pareto distribution [10]. Since CPU processing is impacted by newly arriving flows, we marked a subset of these flows as new flows in order to exercise the CPU resource constraint [10]. We run each experiment multiple times with different random seeds to get a stable mean and standard deviation.

## 4.2 Resource Usage and Traffic Trade-off

The goal of vCRIB rule placement is to minimize the traffic overhead given the resource constraints. To calibrate vCRIB's performance, we compare it against *SourcePlacement*, which stores the rules at the source hypervisor. Our metric for the efficacy of vCRIB's performance is the ratio of traffic as a result of vCRIB's




(a) Memory budget at servers (b) CPU budget at servers

**Figure 8:** *Traffic overhead and resource constraints tradeoffs*

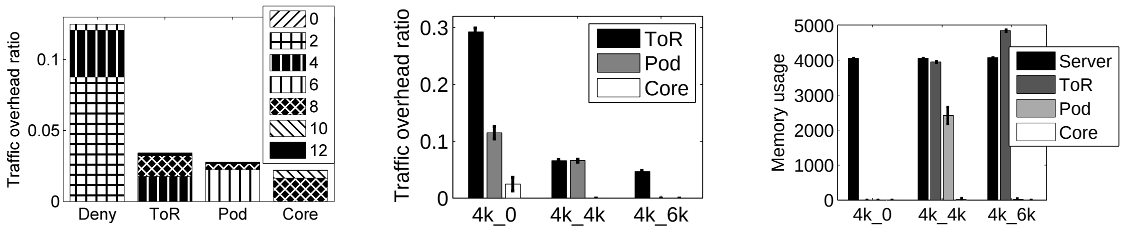
rule placement to the traffic incurred as a result of SourcePlacement (regardless of whether SourcePlacement is feasible or not). When *all* the servers have enough capacity to process rules (i.e., SourcePlacement is feasible), it incurs lowest traffic overhead; in these cases, vCRIB automatically picks the same rule placement as SourcePlacement, so here we only evaluate cases that SourcePlacement is infeasible. We begin with memory resource model at servers because of its simpler similarity model and later compare it with CPU-constrained servers.

**vCRIB uses similarity to find feasible solutions when SourcePlacement is infeasible.** With *Range* IP allocation, partitions in the Source IP dimension which are similar to each other are saved on one server, so the average load on machines is smaller for SourcePlacement. However, there may still be a few overloaded machines that result in an infeasible SourcePlacement. With *Random* IP allocation, the partitions on a server have low similarity and as a result the average load of machines is larger and there are many overloaded ones. Having the maximum load of machines above 5K in all runs for both Range and Random cases, we set a capacity of 4K for servers and 0 for switches ("4K_0" setting) to make SourcePlacement infeasible. vCRIB could successfully fit all the rules in the servers by leveraging the similarities of partitions and balancing the rules. The power of leveraging similarity is evident when we observe that in the Random case *the average number of rules per machine (4.2K) for SourcePlacement exceeds the server capacity*, yet vCRIB finds a feasible placement by saving similar partitions on the same machine. Moreover, vCRIB finds a feasible solution when we add switch capacity and uses this capacity to optimize traffic (see below), yet SourcePlacement is unable to offload the load.

**vCRIB finds a placement with low traffic overhead.** Figure 8(a) shows the traffic ratio between vCRIB and SourcePlacement for the Range and Random cases with error bars representing standard deviation for 10 runs. For the Range IP assignment, vCRIB minimizes the traffic overhead under 0.1%. The worst-case traffic overhead for vCRIB is 21% when vCRIB cannot leverage rule processing in switches to place rules and the VM IP address allocation is random, an adversarial setting for vCRIB. The reason is that in the Random case the arrangement of the traffic sources is oblivious to the similarity of partitions. So any feasible placement depending on similarity puts partitions far from their sources and incurs traffic overhead. When it is possible to process rules on switches, *vCRIB's traffic overhead decreases dramatically* (6% (3%) for 4K (6K) rule capacity in internal switches); in these cases, to meet resource constraints, vCRIB places partitions on ToR switches on the path of traffic, incurring minimal overhead. As an aside, these results illustrate the potential for using vCRIB's algorithms for *provisioning*: a data center operator might decide when, and how much, to add switch rule processing resources by exploring the trade-off between traffic and resource usage.

**vCRIB can also optimize placement given CPU constraints.** We now consider the case where servers may be constrained by CPU allocated for rule processing (Figure 8(b)). We vary the CPU budget allocated to rule processing (10%, 20%, 40%) in combination with zero, 4K or 6K memory at switches. For example in case "40_0" (i.e., each server has 40% CPU budget, but there is no capacity at switches), SourcePlacement results in an infeasible solution, since the highest CPU usage is 56% for range IP allocation and 42% for random IP allocation. In contrast, vCRIB can find feasible solutions in all the cases except "10_0" case. When we have only 10% CPU budget at servers, vCRIB needs some memory space at the switches (e.g., 4K rules) to find a feasible solution. With a 20% CPU budget, vCRIB can find a feasible solution even without any switch capacity ("20_0"). With higher CPU budgets, or with additional switch memory, vCRIB's traffic overhead becomes negligible. Thus, vCRIB can effectively manage heterogeneous resource constraints and find low traffic-overhead placement in these settings. Unlike with memory constraints, Range IP assignment with CPU constraints does not have a lower average load on servers for SourcePlacement, nor does it have a feasible solution with lower traffic overhead, since with the CPU resource usage function closer partitions in the source IP dimension are no longer the most similar.

**vCRIB can handle switch only cases.** There are scenarios where the operator cannot save the rules at hypervisors. For example, the rules need a special hardware that is only available at switches, or the operator does not want to install hypervisors at servers as it has only one VM per machine. vCRIB works in this case as good as the previous cases owing to the fact that it can adapt the placement of rules based on the available resources in the data center. For example, if it is not possible to place all rules for VMs of a rack on ToR switch, vCRIB saves the rules on other ToR switches or other internal switches and puts only the forwarding rules on the servers. The




(a) Traffic overhead for different rules (b) Traffic overhead on different links (c) Memory usage on different devices

**Figure 9:** *Spatial distribution of traffic and resource usage*

Figure 10 shows the traffic overhead vCRIB introduces in Range and Random IP assignment cases where putting the rules on ToR switches needs 7951 (5054) and 6974 (5564) rules in maximum (mean) respectively. vCRIB could find a feasible solution for 0K_4K case, while 39% more traffic comparing to saving rules on ToR switches may be unacceptable. However, vCRIB shows that to handle the current ruleset there is no need to upgrade switches to 8*K* rules capacity but only 6*K* rules capacity is enough thanks to vCRIB which finds the optimized placement with only 0.1% traffic overhead for 0K_6K.

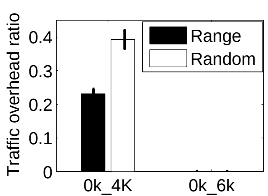



**Figure 10:** *Traffic overhead ratio for switch only cases*

**vCRIB's integrated rule management is beneficial.** We explore the benefits of vCRIB's ability to jointly manage tenant rules. Suppose we have two tenants in the cloud. Each tenant has 10 of the 20 VMs on each server, its own IP ranges (half of the IP space), defines its own rules on the traffic among its own VMs. Each tenant also generates 500 out of 1K flows of each server. Assume server memory constraint is 4K and switches have no rule processing capacity. We compare two settings: (1) *Disjoint:* Each tenant has half of memory (2K) to run vCRIB independently (2) *Joint:* A single vCRIB controller jointly manage rules for both tenants. Although the disjoint setting already leverages vCRIB to fully leverage its allocated resources and there is no overlapping across the two sets of rules, the disjoint setting has 0.5% more traffic than the Joint setting. This is because the joint setting can balance the rules across tenants at different servers. For example, if all the flows from one VM of Tenant A should be accepted, the joint setting can save these accept rules at the destination server leaving more space for the rules of Tenant B.

### 4.3 Resource Usage and Traffic Spatial Distribution

We now study how resource usage and traffic overhead are spatially distributed across a data center for the Random case.

**vCRIB is effective in leveraging on-path and nearby devices.** Figure 9(a) shows the case where servers have a capacity of 4K and switches have none. We classify the rules into deny rules, accept rules whose traffic stays within the rack (labelled as "ToR"), within the Pod ("Pod"), or goes through the core routers ("Core"). In general, vCRIB may redirect traffic to other locations away from the original paths, causing traffic overhead. We thus classify the traffic overhead based on the hops the traffic incurs, and then normalize the overhead based on the traffic volume in the SourcePlacement approach. Adding the percentage of traffic that is handled in the same rack of the source for deny traffic (8.8%) and source or destination for accept traffic (1.8% ToR, 2.2% POD, and 1.6% Core), shows that out of 21% traffic overhead, about 14.4% is handled in nearby servers.

**Most traffic overhead vCRIB introduces is within the rack.** Figure 9(b) classifies the locations of the extra traffic vCRIB introduces. vCRIB does not require additional bandwidth resources at the core links; this is advantageous, since core links can limit bisection bandwidth. In part, this can be explained by the fact that only 20% of our traffic traverses core links. However, it can also be explained by the fact that vCRIB places partitions only on ToRs or servers close to the source or destination. For example, in the "4K_0" case, there is 29% traffic overhead in the rack, 11% in the Pod and 2% in the core routers, and based on Figure 9(c) all partitions are saved on servers. However, if we add 4K capacity to internal switches, vCRIB will offload some partitions to switches close to the traffic path to lower the traffic overhead. In this case, for accept rules, the ToR switch is on the path of traffic and does not increase traffic overhead. Next, 4k_6k case, in Figure 9(c), shows that if we increase the capacity of ToR switches, we can compact all rules saved on internal switches on them and decrease the traffic overhead more. The interesting point is that 4k_6k uses less resources (computed by summing the bars) comparing to 4k_4k as it can compact more partitions by leveraging their similarity. Note that the servers are always full as they are the best place for saving partitions.

### 4.4 Parameter Sensitivity Analysis

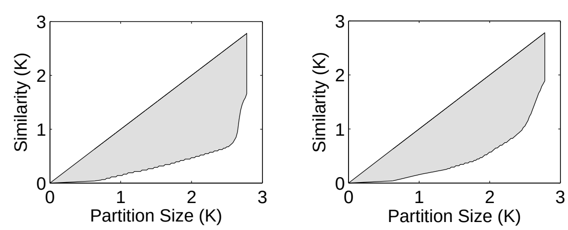
The IP assignment method, traffic locality and rules in partitions can affect vCRIB performance in finding a feasible solution with low traffic. Our previous evaluations have explored *uniform* IP assignment for two extreme cases Range and Random above. In this section, we evaluate the performance of vCRIB for skewed distribution of the number of IPs/VMS per machine, different machine capacities, different traffic locality patterns, and partitions with different sizes and similarities.

**vCRIB is effective for highly skewed rule distribution across servers.** Since data center VM placement is dictated by considerations other than rule processing, it may be possible for SourcePlacement to result in a *skewed* distribution of rules on servers: i.e., some servers have a large number of rules for example because the VMs placed on them source many flows. To mimic skewed rule distributions, we vary the distribution of #VMs at different servers using a Pareto distribution (with mean 20 and $k = 3$). Compared with the uniform distribution of #VMs, those servers with more VMs have more rules accordingly, and need vCRIB to balance the load. In these cases, vCRIB is able to balance rules across servers, and thus incurs low traffic overhead. vCRIB uses this non-uniformity and puts partitions on under-loaded machines near overloaded ones.

**vCRIB is effective for different device capacities.** We compare the performance of FFDS algorithm with Greedy Packing algorithm (Algorithm 2) by evaluating the number of used devices when we increase the capacity of devices from the size of the largest partition to the size of all rules in Figure 11. Besides evaluating for ClassBench rules (Figure 11(a)), we tested the algorithms for Random rules too (Figure 11(b)). The ranges for these rules are selected based on uniform distribution while making sure that they are prefix wildcard. Firstly, the diagrams show that the performance of Greedy Packing algorithm is within a factor of 2 of the bound for Fractional Packing algorithm. Secondly, it confirms that FFDS works better than Greedy Packing algorithm. Lastly, it shows that selecting the largest partition first (FFDS LF) slightly improves the performance of FFDS.

**vCRIB has lower traffic overhead for less local traffic patterns.** We explored the effect of traffic locality on vCRIB traffic overhead in Figure 11(c) where the numbers for each series represent the probability of connections inside a rack (ToRP), inside a Pod (PodP) and through the core switches (CorP) respectively for each source server. The result shows that having more non-local flows gives vCRIB more choices to offload rule processing and reach feasible solutions with lower traffic overhead.

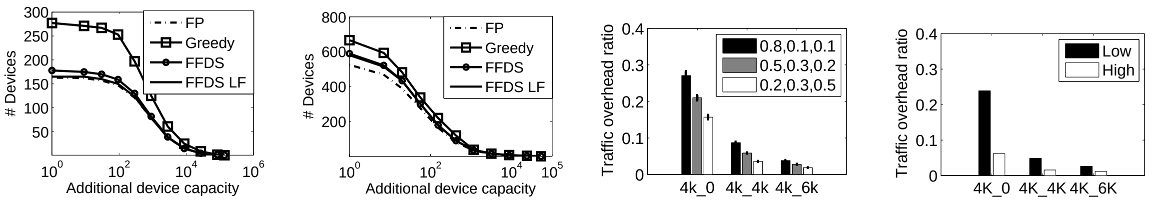
**vCRIB uses similarity to accommodate larger partitions.** We have explored two properties of the rules in partitions by changing the ruleset. In Figure 12, we define a two dimensional space: one dimension measures the average similarity between partitions and the other the average size of partitions. Intuitively, the size of partitions is a measure of the difficulty in finding a feasible solution and similarity is the property of a ruleset that vCRIB exploits to find solutions. To generate this figure, we start from an infeasible setting for SourcePlacement with a maximum of 5.7K rules for "4k_0" setting and then change the ruleset without changing the load on the maximum loaded server. We then explore the two dimensions as follows. Starting from the ClassBench ruleset and Range IP assignment, we split rules into half in the source IP dimension to decrease similarity without changing partition sizes. To increase similarity, we extend a rule in source IP dimension and remove rules in the extended area to maintain the same partition size. Adding or removing rules matching only one VM (micro rules), also help us change average partitions size without changing the similarity. Unfortunately, removing just micro rules is not enough to explore the entire range of partition sizes, so we also remove rules randomly.




(a) Feasibility region (b) 10% traffic overhead

**Figure 12:** *vCRIB working region and ruleset properties*

Figure 12(a) presents the *feasibility region* for vCRIB regardless of traffic overhead. Since average similarity cannot be more than the average partition size, the interesting part of the space is below the 45°. Note that vCRIB is able to cover a large part of the space. Moreover, the shape of the feasibility region shows that for a fixed average partition size, vCRIB works better for partitions with larger similarity. This means that to handle larger partitions, vCRIB needs more similarity between partitions; however, this relation is not linear since vCRIB may not be able to utilize the available similarity given limits on server capacity. When considering only solutions with less than 10% traffic overhead, vCRIB's




(a) Classbench rules (b) Random rules (c) Traffic locality (d) Diff. similarity of partitions

**Figure 11:** *Parameter sensivity analysis for traffic patterns, machine sizes and similarity of partitions*

feasibility region (Figure 12(b)) is only slightly smaller. This figure demonstrates vCRIB's utility: for a small additional traffic overhead, vCRIB can find many additional operating points in a data center that, in many cases, might have otherwise been infeasible.

We also tried a different method for exploring the space, by tuning the IP selection method on a fixed ruleset, and obtained qualitatively similar results. To control the similarity of partitions, we adapted our IP address allocation method to tunably control the similarity of partitions assigned to a server in VM placement. Instead of selecting IPs uniformly from the ranges found by rule boundaries, we define a Normal distribution on the source IP dimension. As a result, because of the distribution mass in the middle of the IP range, IP addresses will be closer to each other and more similar. Figure 11(d) shows that the traffic overhead of vCRIB is much lower with high similarity across partitions.

### 4.5 Traffic vs. Resource Usage Trend

We draw the trend of traffic ratio and resource utilization of the data center during the *batch* traffic-refinement for memory model and CPU model case in Figure 13. Each point represents 1000 partition migration steps and the curve starts from the output of resource-aware placement with high traffic and goes to an optimized low traffic placement (top-down). The resource-aware placement algorithm compacts the partitions into a few devices, so there may be many under-loaded devices in the early steps of the traffic-refinement. Therefore, migrating the partitions unpacks the solution of resource-aware placement in the early steps. That is why in Figure 13(a), each line goes to the right direction fast. However, as the resources around the sources are being used up, migration steps just swap partitions, so the resource usage will be almost the same but the traffic overhead will decrease. Besides, when we add capacity to internal switches, the resource utilization decreases. Part of this decrease is because the rules are compacted into the larger ToR switches and Figure 9(c) confirms that.

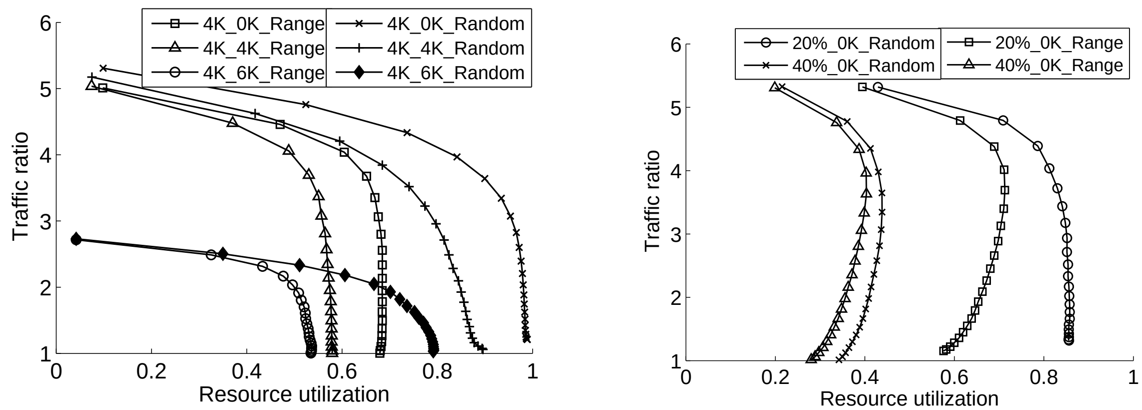
The CPU model diagram in Figure 13(b) also shows the same relation between Random vs Range IP assignment schemes and low vs high resources availability. However, the interesting point is that the curves come back to left at the end which means the resource usage decreases even when we are optimizing the traffic. We believe that this is because of the resource usage required for handling new flows. If we do not save a partition on the source server, *two* machines need to handle all of its new flows: source and current host. However, as vCRIB moves partitions to their source servers (or even destination of accepted flows), the resource usage to handle the partition decreases. In the first few thousand migration steps, the resource usage increases as moving partitions to empty sources increases the number of wild card and rules and its effect is larger than handling flows on sources. Then the reduction of new flows overhead overcomes the effect of wild card and rules. Because similar rules and wildcards will not add any overhead to the devices, but there is no similarity relation among partitions based on new flows.

### 4.6 Reaction to Cloud Dynamics

Figure 14 compares benefit-greedy (with timeout 10 seconds) with overhead-greedy and a randomized algorithm[9] after a single VM migration for the 4K_0 case. Each point in Figure 14 shows a step in which one partition is moved, and the horizontal axis is time in log scale. At time A, we migrate a VM from its current server $S_{old}$ to a new one $S_{new}$, but $S_{new}$ does not have any space for the partition of the VM, $P$. As a result, $P$ remains on $S_{old}$ and the traffic overhead increases by $40MBps$. Both benefit-greedy and overhead-greedy move the partition $P$ for the migrated VM to a server in the rack containing $S_{new}$ at time B and reduce traffic by 20Mbps. At time B, benefit-greedy brings out two partitions from their current host $S_{new}$ to free up the memory for $P$ while imposing a little traffic overhead. At time C, benefit-greedy moves $P$ to $S_{new}$ and reduces traffic further by $15Mbps$. The entire process takes only 5 seconds. In contrast, the randomized algorithm takes 100 seconds to find the right partitions and thus is not useful with these dynamics.

We then run multiple VM migrations to study the average behavior of benefit-greedy with 5 and 10 seconds timeout. In each 20 seconds interval, we randomly pick a VM and move it to another random server. Our sim-

[9] Markov Approximation [22] with target switch selection probability $\propto$ exp(traffic reduction of migration step)




(a) Memory model

(b) CPU model

**Figure 13:** *Traffic ratio vs resource utilization ratio during batch traffic-refinement*

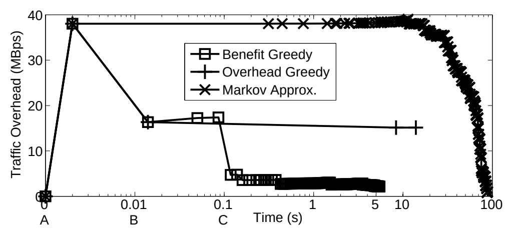



**Figure 14:** *Traffic refinement for one VM migration*

ulations last for 30 minutes. The trend of data center traffic in Figure 15 shows that benefit-greedy maintains traffic levels, while overhead-greedy is unable to do so. Over time, benefit-greedy (both configurations) reduces the average traffic overhead around 34 MBps, while overhead-greedy algorithm increases the overhead by 117.3 MBps. Besides, this difference increases as the interval between two VM migration increases.

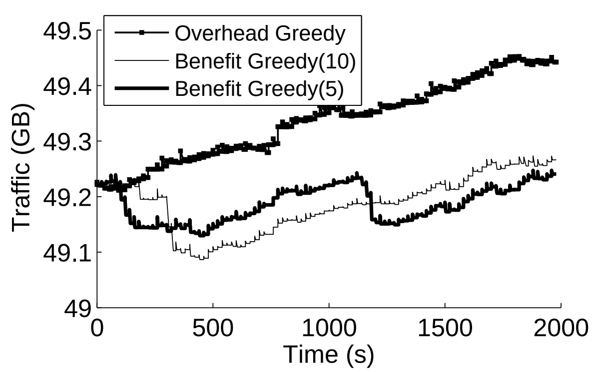



**Figure 15:** *The trend of traffic during multiple VM migration*

### 4.7 Prototype Evaluation

We built vCRIB prototype using Open vSwitch [4] as servers and switches, and POX [1] as the platform for vCRIB controller for micro-benchmarking.

**Overhead of collecting traffic information:** In our prototype, we send traffic information collected from each server's Open vSwitch kernel module to the controller. Each piece of information requires 13 Bytes for 5 tuples[10] and 2 Bytes for the traffic change volume.

Since we only need to detect traffic changes at the rule-level, we can more aggressively filter the traffic information than traditional traffic engineering solutions [11]. The vCRIB controller sets a threshold $\delta(F)$ for traffic changes of a set of flows $F$ and sends the threshold to the servers. The servers then only report traffic changes above $\delta(F)$. We set the threshold $\delta$ for two different granularities of flow sets $F$. A larger set $F$ makes vCRIB less sensitive to individual flow changes and leads to less reporting overhead but incurs less accuracy. (1) We set $F$ as the volume each *rule* for each *destination server* in each *per-source partition*. (2) We assume all the rules in a partition have accept actions (as the worst case for traffic). Thus, the vCRIB controller sets the threshold that affects the size of traffic to each *destination server* for each *per-source partition* (summing up all the rules). If there are 20 flow changes above the threshold, we need to send 260B/s per server, which means 20Mbps for 10K servers in the data center. For VM migrations and rule insertion/deletion, the vCRIB controller can be notified directly by the the data center management system.

**Controller overhead:** We measure the delay of processing 200K ClassBench rules. Initially, the vCRIB controller partitions these rules, runs the resource-aware placement algorithm and the traffic-aware refinement to derive an initial placement; this takes up to *five* minutes. However, these recomputations are triggered only when a placement becomes infeasible; this can happen after a long sequence of rule changes or VM add/remove.

The traffic overhead of rule installation and removal depends on the number of refinement steps and the num-

[10]Some rules may have more packet header fields and thus require more bytes. In this cases, we can compress these information using fingerprints to reduce the overhead.

ber of rules per partition. The size of OpenFlow command for a rule entry is 100 Bytes, so if a partition has 1K rules, the overhead of removing it from one device and installing at another device is 200KB. For each VM migration, which needs an average of 11 partitions, the bandwidth overhead of moving the rules is 11×200KB=2.2MB.

**Reaction to cloud dynamics:** We evaluate the latency of handling traffic changes by deploying our prototype in a topology with five switches and six servers as shown in Figure 1. We deploy a vCRIB controller that connects with all the devices with an RTT of 20 ms. We set the capacity of each server/switch as large enough to store at most one partition. We then inject a traffic change pattern that causes vCRIB to swap two partitions and add a redirection rule at a VM. It takes vCRIB *30ms* to detect the traffic changes, and move the rules to the new locations.

# 5 Related Work

Our work is inspired by several different strands of research, each of which we cover briefly.

**Policies and rules in the cloud:** Recent proposals for new policies often propose customized systems to manage rules on either hypervisors [4, 13, 33, 31]) or switches [3, 8, 30]. vCRIB proposes an abstraction of a centralized rule repository for all the policies, frees these systems from the complexity inherent in the rule management, and handles heterogeneous resource constraints at devices while minimizing the traffic overhead.

**Rule management in software-defined networks (SDNs):** Recent work on SDNs provides rule repository abstractions and some rule management capabilities [12, 25, 39, 13]. vCRIB focuses on data centers, which are more dynamic, more sensitive to traffic overhead, and face heterogeneous resource constraints.

**Distributed firewall:** Distributed firewalls [9, 21], often used in enterprises, leverage a centralized manager to deploy security policies on edge machines. vCRIB manages more fine-grained rules on flows and VMs for various policies including firewalls in the cloud. Rather than placing these rules at the edge, vCRIB places these rules taking into account the rule processing constraints, while minimizing traffic overhead.

**Rule partition and placement solutions:** The problem of partitioning and placing multi-dimensional data at different locations also appears in other contexts. Unlike traditional partitioning algorithms [37, 35, 17, 27, 26] which divide rules into partitions using a top-down approach, vCRIB uses *per-source partitions* to place the partitions close to the source with low traffic overhead. Compared with DIFANE [39], which *randomly* places *a single* partition of rules at each switch, vCRIB takes the *partitions-with-replication* approach to flexibly place *multiple* per-source partitions at one device. In preliminary work [28], we proposed an *offline* placement solution which works *only* for the TCAM resource model. The paper has a top-down heuristic partition-with-split algorithm which cannot limit the overhead of redirection rules and is not optimized for CPU-based resource model. Besides, having partitions with traffic from multiple sources requires complicated partition replication to minimize traffic overhead. In contrast, vCRIB uses fast per-source partition-with-replication algorithm which reduces TCAM-usage by leveraging similarity of partitions and restricts the resource usage of redirection by using limited number of equal shaped redirection rules. Our preliminary work used an unscalable DFS branch-and-bound approach to find a feasible solution and optimized the traffic in one step. vCRIB scales better using a two-phase solution where the first phase has an approximation bound in finding a feasible solution and the second can be run separately when the placement is still feasible.

# 6 Conclusion

vCRIB, is a system for automatically managing the fine-grained rules for various management policies in data centers. It jointly optimizes resource usage at both switches and hypervisors while minimizing traffic overhead and quickly adapts to cloud dynamics such as traffic changes and VM migrations. We have validated its design using simulations for large ClassBench rulesets and evaluation on a vCRIB prototype built on Open vSwitch. Our results show that vCRIB can find feasible placements in most cases with very low additional traffic overhead, and its algorithms react quickly to dynamics.

# References


[1] `http://www.noxrepo.org/pox/about-pox`.

[2] `http://www.praxicom.com/2008/04/the-amazon-ec2.html`.

[3] Big Switch Networks. `http://www.bigswitch.com/`.

[4] Open vSwitch. `http://openvswitch.org/`.

[5] Private conversation with Amazon.

[6] Private conversation with rackspace operators.

[7] Virtual networking technologies at the server-network edge. `http://h20000.www2.hp.com/bc/docs/support/SupportManual/c02044591/c02044591.pdf`.

[8] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat. Hedera: Dynamic Flow

Scheduling for Data Center Networks. In *NSDI*, 2010.

[9] S. M. Bellovin. Distributed Firewalls. *;login:*, November 1999.

[10] T. Benson, A. Akella, and D. A. Maltz. Network Traffic Characteristics of Data Centers in the Wild. In *IMC*, 2010.

[11] T. Benson, A. Anand, A. Akella, and M. Zhang. MicroTE: Fine Grained Traffic Engineering for Data Centers. In *ACM CoNEXT*, 2011.

[12] M. Casado, M. Freedman, J. Pettit, J. Luo, N. Gude, N. McKeown, and S. Shenker. Rethinking Enterprise Network Control. *IEEE/ACM Transactions on Networking*, 17(4), 2009.

[13] M. Casado, T. Koponen, R. Ramanathan, and S. Shenker. Virtualizing the Network Forwarding Plane. In *PRESTO*, 2010.

[14] C. Chekuri and S. Khanna. A PTAS for the Multiple Knapsack Problem. In *SODA*, 2001.

[15] R. Cohen, L. Katzir, and D. Raz. An Efficient Approximation for the Generalized Assignment Problem. *Information Processing Letter*, 100(4), 2006.

[16] A. Curtis, J. Mogul, J. Tourrilhes, P. Yalagandula, P. Sharma, and S. Banerjee. DevoFlow: Scaling Flow Management for High-Performance Networks. In *SIGCOMM*, 2011.

[17] P. Gupta and N. McKeown. Packet Classification using Hierarchical Intelligent Cuttings. In *Hot Interconnects VII*, 1999.

[18] M. Hajiaghayi, K. Jain, K. Konwar, L. Lau, I. Mandoiu, A. Russell, A. Shvartsman, and V. Vazirani. The Minimum k-Colored Subgraph Problem in Haplotyping and DNA Primer Selection. In *IWBRA*, 2006.

[19] B. Heller, S. Seetharaman, P. Mahadevan, Y. Yiakoumis, P. Sharma, S. Bannerjee, and N. McKeown. ElasticTree: Saving Energy in Data Center Networks. In *NSDI*, 2010.

[20] T. L. Hinrichs, N. S. Gude, M. Casado, J. C. Mitchell, and S. Shenker. Practical Declarative Network Management. In *WREN*, 2009.

[21] S. Ioannidis, A. D. Keromytis, S. M. Bellovin, and J. M. Smith. Implementing a Distributed Firewall. In *CCS*, 2000.

[22] J. Jiang, T. Lan, S. Ha, M. Chen, and M. Chiang. Joint VM Placement and Routing for Data Center Traffic Engineering. In *INFOCOM*, 2012.

[23] E. G. C. Jr., M. R. Carey, and D. S. Johnson. Approximation Algorithms for NP-hard Problems. chapter Approximation Algorithms for Bin Packing: A Survey. PWS Publishing Co., Boston, MA, USA, 1997.

[24] S. Kandula, S. Sengupta, A. Greenberg, P. Patel, and R. Chaiken. The Nature of Datacenter Traffic: Measurements and Analysis. In *IMC*, 2009.

[25] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, and S. Shenker. Onix: A Distributed Control Platform for Large-scale Production Networks. In *OSDI*, 2010.

[26] V. Kriakov, A. Delis, and G. Kollios. Management of Highly Dynamic Multidimensional Data in a Cluster of Workstations. *Advances in Database Technology-EDBT*, 2004.

[27] A. Mondal, M. Kitsuregawa, B. C. Ooi, and K. L. Tan. R-tree-based Data Migration and Self-Tuning Strategies in Shared-Nothing Spatial Databases. In *GIS*, 2001.

[28] M. Moshref, M. Yu, A. Sharma, and R. Govindan. vCRIB: Virtualized Rule Management in the Cloud. In *HotCloud*, 2012.

[29] J. Mudigonda, P. Yalagandula, J. Mogul, and B. Stiekes. NetLord: A Scalable Multi-Tenant Network Architecture for Virtualized Datacenters. In *SIGCOMM*, 2011.

[30] L. Popa, A. Krishnamurthy, S. Ratnasamy, and I. Stoica. FairCloud: Sharing The Network In Cloud Computing. In *HotNets*, 2011.

[31] L. Popa, M. Yu, S. Y. Ko, I. Stoica, and S. Ratnasamy. CloudPolice: Taking Access Control out of the Network. In *HotNets*, 2010.

[32] S. Rupley. Eyeing the Cloud, VMware Looks to Double Down On Virtualization Efficiency, 2010. `http://gigaom.com/2010/01/27/eyeing-the-cloud-vmware-looks-to-double-down-on-virtualization-efficiency`.

[33] A. Shieh, S. Kandula, A. Greenberg, C. Kim, and B. Saha. Sharing the Datacenter Networks. In *NSDI*, 2011.

[34] M. Sindelar, R. K. Sitaram, and P. Shenoy. Sharing-Aware Algorithms for Virtual Machine Colocation. In *SPAA*, 2011.

[35] S. Singh, F. Baboescu, G. Varghese, and J. Wang. Packet Classification Using Multidimensional Cutting. In *SIGCOMM*, 2003.

[36] D. E. Taylor and J. S. Turner. ClassBench: A Packet Classification Benchmark. *IEEE/ACM Transactions on Networking*, 15(3), 2007.

[37] B. Vamanan, G. Voskuilen, and T. N. Vijaykumar. Efficuts: Optimizing Packet Classification for Memory and Throughput. In *SIGCOMM*, 2010.

[38] A. Voellmy, H. Kim, and N. Feamster. Procera: A Language for High-Level Reactive Network Control. In *HotSDN*, 2010.

[39] M. Yu, J. Rexford, M. J. Freedman, and J. Wang. Scalable Flow-Based Networking with DIFANE. In *SIGCOMM*, 2010.