RAMPS

REAL-TIME APPLICATION MEASUREMENT and PLACEMENT SYSTEM

Pratheek Nagaraj, Manali Naik, Jenny Shen
pnagaraj@mit.edu, mnaik@mit.edu, jenshen@mit.edu

May 9, 2014

6.033 Design Project 2

Report

1 Introduction

With large scale data production and accumulation, processing data now often relies on networks of computation cores. Many large computational tasks are now commonly run in public data centers like Amazon and Rackspace. Users divide their tasks into smaller jobs that can be deployed onto virtual machines (VMs) in those data centers. However, the increasing demand for VMs can lead to high network traffic, significantly impacting the performance of user applications. RAMPS, the Real-time Application Measurement and Placement System, is a system designed to optimize cost for users running computations in large data center networks in the presence of these challenges. It reduces the computational time of VMs by taking periodic measurements of data transfer rates and group occupancy to adapt to network conditions.

2 Overview

With RAMPS, there are four different types of VMs in the system, each with various responsibilities: the CzarVM, SerfVMs, DukeVMs, and PeasantVMs (Table 1). The CzarVM and SerfVMs are additional VMs (beyond the n required), while DukeVMs and PeasantVMs are workers that perform application computations. To limit overhead cost, RAMPS uses one CzarVM to make centralized placement decisions, and spawns SerfVMs only when exploring new areas of the network. Throughput and latency measurements are taken in a distributed fashion by DukeVMs and PeasantVMs. Finally, RAMPS uses two different kinds of VM movement: single transfer and bulk transfer. The CzarVM determines which to use depending on network conditions and current VM performance.

VM Type	Actions	Count	Auctioning	Application Computation
CzarVM	Central decision maker	1	✓	
SerfVM	Measures performance in unused groups	$0 \sim 10$		
DukeVM	Performs computation, relays information to CzarVM	n total, combined with PeasantVMs	✓	✓
PeasantVM	Performs computation			✓

Table 1: Four types of VMs in RAMPS. Actions describe the roles of each VM type, count indicates the number of each type of VM active, auctioning indicates whether the role can be transferred to another VM, and application computation indicates whether the data transferred directly affect the progress of the application.

Section 3 describes RAMPS's overall design, including initialization of the system, network measurements, and VM movement. Section 3.4 also includes discussion of potential optimizations while Section 3.5 discusses design alternatives. Section 4 analyzes performance, examines specific use cases, and describes RAMPS's limitations.

3 Design

RAMPS is built on top of a data center network that uses 1,152 physical machines partitioned into groups of 48, each with the ability to host 4 VMs. The links between machines have a data capacity of 10 gigabits per second. Since cost on the network is measured by VM usage per minute, the design balances exploration and exploitation of VM movement to reduce VM usage time.

3.1 Initialization

When a computation begins, the first VM initialized is designated the CzarVM, or the core VM of the system. Using the function `random_place(v)`, the CzarVM is randomly placed on a machine. The CzarVM stores the following data structures with information about the system:

- `GROUP_OCCUPANCIES`
  A table of group numbers mapping to the number of machine vacancies in their group.
- `P_ADDRESSES`
  A table of the computations VMs mapping to their physical machine addresses.
- `SCORES`
  A priority queue ranking the VMs by their calculated scores (Section 3.2.4).
- `IP_ADDRESSES`
  A table of the computations VMs mapping to their IP addresses.

To begin placing the computation's virtual machine jobs, we use B, the given $n \times n$ matrix specifying the number of bytes each pair of virtual machines will transfer, to determine an ordering. The VMs will be ordered according to this ranking algorithm:

```
Create a traffic graph G(V,E), where the nodes are VMs and edge weights are the
     corresponding values in B
Sort E by decreasing edge weight
Build a tree from the node with the heaviest out-edge, choosing edge weights
     greedily to maximize the total weight of tree
```

The order the algorithm determines attempts to place the largest computational tasks close together in the same group to reduce VM usage time.

The CzarVM will then measure how many open VM slots each group of machines has, and orders the groups of machines by decreasing number of the group's open slots. To begin, the VMs are placed on the machines in the group with the greatest number of open slots. These placed VMs are PeasantVMs that perform user computation.

As the virtual machines are placed, the CzarVM designates a DukeVM as a group leader for every group in the network hosting PeasantVMs; these leaders are also PeasantVMs and do not require extra cost (Figure 1). Each DukeVM obtains and relays the group machine occupancy to the CzarVM every second. It does so by calling `machine_occupancy(m)` on all physical machines in its group every second, allowing it to quickly detect openings. The CzarVM updates `GROUP_OCCUPANCIES` with these occupancies. A DukeVM has a *term length* of 10s. Upon the ninth second of its term, it sends requests for the scores of each VM (Section 3.2.4) in its group. It then auctions off its role to the highest-scoring VM in the group, transferring its occupancy data to the new leader [1]. This is to ensure that the system avoids slow-performing components and failure modes.

To ensure fast data transfer between the CzarVM and the DukeVMs, the CzarVM has a term length of 15s. When its term is over, the CzarVM calculates the average VM score (Section 3.2.4) of each group hosting PeasantVMs, and it moves to an empty slot in the group with the highest average score. All DukeVMs are notified of the CzarVM's new location, and they relay this information to individual VMs.

Although this initialization method requires extra computation time before any PeasantVMs are placed, it allows the application to achieve a more optimal configuration from the beginning. Random initial placement could result in poor performance that would require several VM movements over multiple seconds. RAMPS makes a trade-off for better initial performance at the cost of spending time doing preliminary computations on the CzarVM.

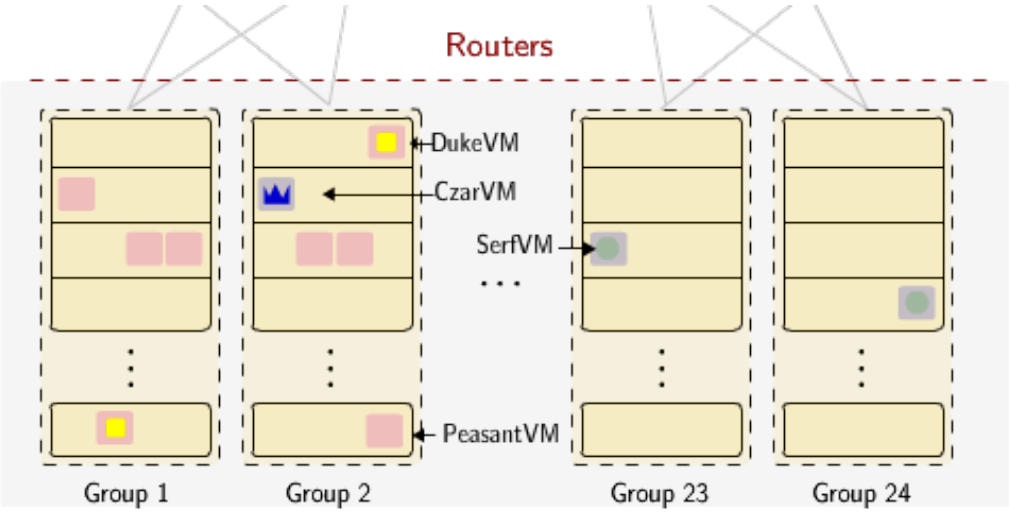

Figure 1: Diagram depicting the placement of the 4 types of VMs on the network. Each group performing computation has one DukeVM, which is also a PeasantVM, denoted in red. There is one CzarVM (randomly placed), and SerfVMs will explore unused groups to measure performance.

3.2 Measurements

Each PeasantVM uses a combination of active and passive probing to measure performance on each of its outgoing TCP connections. Incoming connections are ignored to avoid double-counting the same measurements on two different VMs. Since each VM attempts to reduce cost on its outgoing connections, the overall performance of all VMs approaches the optimal solution.

This combination of probing is used to calculate two metrics: realized throughput and latency (round-trip time). Realized throughput is different from the value returned by the `tcp_throughput` function in that it represents the throughput on the connection while the VM was actively sending data. By ignoring the time when no data was transmitted, the system obtains more accurate measurements of throughput on the connection.

3.2.1 Realized Throughput

To calculate realized throughput of each outgoing connection, VMs maintain a *transmission history* for each connection - a collection of timestamps marking the times intervals when it had outstanding packets sent to another worker. The *transmission history* is a dictionary keyed by the destination VM number of each open connection. Once an outgoing connection closes, the key is removed from the dictionary. Each key is mapped to two lists. The first contains the times when the VM sent its first outstanding packet to another worker (when there were none). The second list contains the times when the VM received the ACK for its last outstanding packet, thus ending a period of communication.

The *transmission history* contains timestamps for up to the last 150ms; older records can be discarded. A limit of 150ms is used since the *transmission history* is employed in conjunction with `tcp_throughput`, which only looks at data from the last 100ms. The buffer of 50ms ensures that calculations are correct.

The *transmission history* is used to calculate the *active time*, or the amount of time during the last 100ms that the VM had outstanding packets. This can be calculated by adding up the corresponding intervals from the *transmission history*. Then to calculate $r_{ij,\text{inst}}$, the instantaneous realized throughput from the current VM_i to VM_j, formula 3.1 is used.

$$r_{ij,\text{inst}} = \frac{\texttt{tcp\_throughput(j)} \cdot 0.1}{\text{active\_time}} \tag{3.1}$$

An exponentially-weighted moving average (EWMA) is used to smooth out the values for realized throughput [2]. The resulting average realized throughput, denoted r_{ij}, is given by Equation 3.2. On the first step, the averaged realized throughput is set to be equal to the instantaneous value. The value of ω ($0 < \omega < 1$) is determined empirically.

$$r_{ij}^{(i+1)} = \omega \cdot r_{ij,\text{inst}} + (1 - \omega) \cdot r_{ij}^{(i)} \tag{3.2}$$

3.2.2 Latency

Latency measurements are taken in a similar manner. Each VM maintains a *packet history* recording the times at which individual packets were sent over the network. The *packet history* is also a dictionary mapped by destination VM number. Like the *transmission history*, the dictionary keys only represent open TCP connections. The corresponding values are mappings from sequence numbers to packet transmission timestamps. In the case of packet retransmission, the timestamp is updated to ensure accurate latency measurements. Upon receipt of an ACK, the instantaneous latency, $l_{ij,inst}$, is calculated by subtracting the transmission time (from the *packet history*) from the current time. At this point, the entry in the *packet history* can be removed. Again, EWMA is used to smooth out latency values and θ ($0 < \theta < 1$) is determined empirically (Eq. 3.3).

$$l_{ij}^{(i+1)} = \theta \cdot l_{ij,\text{inst}} + (1 - \theta) \cdot l_{ij}^{(i)} \tag{3.3}$$

Each VM will maintain the smoothed realized throughput and latency values for the duration of each outgoing TCP connection.

3.2.3 Active and Passive Probing

Each VM uses a combination of active and passive probing to adapt to network traffic conditions. Every 1s, VMs runs a probing assessment, a check that uses realized throughput and latency estimates to determine what kind of probing to use on each outgoing connection. When initially placed or moved to a new location, each VM sends $P = 5$ pings packets of default size (32 bytes) on its outgoing connections. This allows the system to make preliminary estimates even when a VM does not transfer any data initially; in the case that it does, the amount of data sent is small enough that it will not significantly impact network conditions. The IP address of the destination is obtained from the TCP packets used in passive probing; if no such packets exist, the VM can attain the address from the CzarVM (in `IP_ADDRESSES`).

The value of P, representing the amount of active probing the VM does in number of ping packets, is adjusted based on performance. Namely, if the current latency estimate l_{ij} is less than L_{max} (determined empirically), then P is increased by 1 ping packet, allowing the VM to do more active probing. However, if l_{ij} reaches L_{max}, then P is halved so less active probing is performed. This adaptive technique uses the additive-increase/multiplicative-decrease model of TCP [3], allowing VMs to dynamically adjust probing on each outgoing connection every 1s. P will also be adjusted to be proportional to realized throughput estimates r_{ij}, so that low-throughput connections will not be overloaded with unnecessary active probing.

3.2.4 Scoring

The system uses a scoring mechanism to estimate overall performance of each outgoing connection on a VM. Each connection leaving VM_i is first assigned an individual subscore using Equation 3.4; higher scores correspond to better performance. Values of parameters α, β, and γ are determined empirically. The Normal CDF (Cumulative Distribution Function) is used to map all subscores from $[0, \infty)$ onto a [0,1] range.

$$\text{subscore}_{ij} = \text{NormCDF}\left[\frac{\alpha \cdot r_{ij,\text{avg}}}{(\beta \cdot \texttt{progress}(i,j) \cdot \gamma \cdot l_{ij,\text{avg}}}\right] \tag{3.4}$$

The subscore_{ij} represents the score on the connection from VM_i to VM_j. Higher subscores represent a better connection between the two VMs, so the subscore increases with larger values of realized throughput. Lower latency connections are preferred, so connections with larger latencies have lower scores. Finally, the subscore also depends on the progress, or the number of bytes that VM_i still has to transmit to VM_j. Lower values of progress mean that the transmission is close to being complete. These receive higher scores because connections that are slow but nearly finished are not as problematic as those that are slow but have much more data left to transfer.

When determining the overall performance of a VM, the system differentiates between outgoing connections that stay within the VM's group and those that leave the group (Figure 2). This distinction is important because connections within a group are always favored to those that cross group boundaries since connections within a group are faster than through routers. The score of a VM - the metric representing its overall performance - uses the subscores from intra-group and inter-group connections in different ways. Equation 3.5 below shows the formula for the score.

$$\text{score}_i = \sum_{\{j\}} (\text{subscore within group})_j - \sum_{\{k\}} (\text{subscore outside group})_k \tag{3.5}$$

This formula not only yields lower scores for VMs with poor individual outgoing connections, but it also penalizes VMs that have several poor-performing inter-group transfers. At the same time, this penalty is offset by the subscores of intra-group connections so that a VM with several good connections within its own group is not moved unnecessarily.

Every 1s, each VM recalculates its subscores/scores and sends a scoring update to the CzarVM. VM_i's update contains two pieces of information:

(1) score_i

(2) destination VM number j such that subscore_{ij} is VM_i's lowest subscore.

These messages are sent in a staggered manner so that all VMs within a group do not try to send them at once. The first machine in each group sends updates to the CzarVM at once, followed by the second machine in each group, etc. The CzarVM uses the messages to update `SCORES` and potentially move a VM to a better location (Section 3.3).

3.3 Movement

In this section we describe the movement procedures used to optimize VM placement. In the steady state, the CzarVM employs a single transfer protocol, whereas in a situation of substantial change (freeing up of VMs), it may use a bulk transfer mechanism. Movement is done by calling the `move` function (pseudocode in Figure 3).

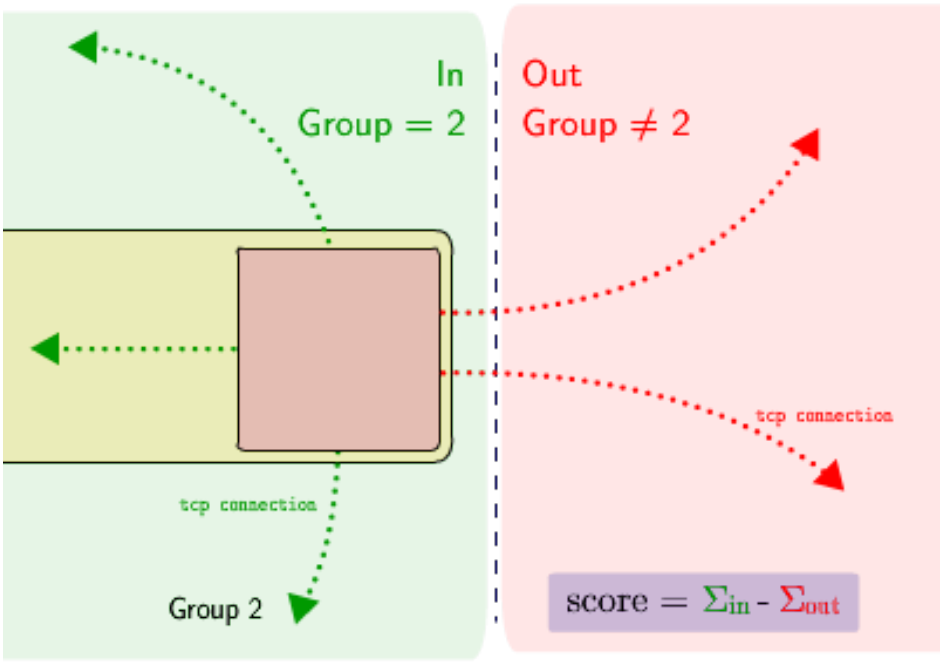

Figure 2: Schematic showing the scoring system for an individual VM. Outgoing TCP connections with the destination in the same group (Group 2, above) are shown in green, while those with destinations outside the group are in red. The score is computed by summing subscores on the green connections, and subtracting the sum of subscores on red connections.

```
# Movement
place(v, m′) # calls acquire_lock(m′, newSlot)
if placement successful:
   remove(v, m) # calls release_lock(m, oldSlot)
```

Figure 3: Pseudocode for the `move` function. m and *oldSlot* are the physical machine and VM slot number (1-4) of the VM's old location. m' and *newSlot* represent the VM's new location. Assume that `place` obtains a lock on the new position and `remove` releases this lock.

3.3.1 Single VM Transfer

Every 1s, the CzarVM finds the mean and standard deviation of the most recent VM scores (stored in `SCORES`), and it identifies VM_{min} with the lowest score overall as a candidate for movement into another group. If the minimum score is more than one standard deviation below the average, the CzarVM moves VM_{min}. To decide where it should be moved, the CzarVM looks at the second entry in its update message the destination corresponding to the VM_{min}'s lowest-subscoring connection. The CzarVM determines which group this destination VM belongs to, and finds open slots in that group. If there are slots open, the CzarVM moves VM_{min} into the new group and updates `P_ADDRESSES` with the new location, otherwise it does nothing.

Upon movement, the VM resets its transmission history, packet history data structures, and estimates for throughput and latency. The VM restarts the procedure in section 3.2 to recalculate these values.

3.3.2 Bulk VM Transfer

The single transfer procedure is good for small scale optimizations on well known metrics, however sometimes we may need to optimize on a larger scale and with less well known information. Specifically, when a large quanity of VMs becomes available either in a single group or across multiple which the user system does not occupy, we need to consider the possible benefit of migrating the computation.

More formally, if the number of empty slots in a single group (not used by the user) or multiple groups (also unused) under the same aggregate router reaches half of group capacity or half the number of VMs needed ($n/2$), then we initiate bulk transfer. This detection is done by the CzarVM who regularly pings each of the groups to obtain the number of empty VMs. Upon detecting this opening, we dispatch SerfVMs that scout the empty slots and more accurately compute a starting realized throughput estimate. The number of SerfVMs dispatched depends on the number of open slots detected and n, the number of VMs needed for computation. This number is low enough such that VMs do not unnecessarily need to be started, and high enough such that we can explore the space quickly (see Table 1).

Finally, if the CzarVM detects that the throughput in the newly opened regions outperform that of the existing computation region(s), the process bulk transfer starts. This involves selecting half of the worst performing VMs as determined by the priority queue the CzarVM holds and shifting them over to the open group(s). This transfer prefers single groups and then those under a single aggregate router. The reason half are chosen is that it takes approximately equal amounts of time to move the remaining VMs to these newly-opened regions and to shift the moved ones back if the bulk transfer is determined to be inefficient. The steady state technique is always working so that any significant anomalies will be smoothed out by the fine-tuning of single VM transfers.

3.4 Optimizations

One optimization is to place a soft fractional threshold on the number of occupied slots allowed for a group. For instance, each group contains 48 machines with 4 VMs each so we can place an 80% cap such that only 150 are freely allowed to be occupied while the remaining can be filled only if the system deems it most efficient.

Another optimization would be to swap pairs of VMs at once. This could prevent situations where VMs take up spots that could be better used by other jobs. While soft group thresholds largely prevent this from occurring, swapping could also solve the issue.

3.5 Design Alternatives

An alternative design considered for placement was using the initial B matrix to define "clusters" of VMs to be placed together at all times. However, RAMPS uses greedy initial placement and relies on single/bulk transfers to move towards an optimal solution. This strategy was chosen because cluster arrangements can change rapidly given fluctuations in VM transfer rates. Instead, RAMPS's scoring system allows it to adapt to these changes quickly, without spending time recomputing cluster boundaries that may no longer reflect system conditions.

An alternative to the measurement component was to permanently place extra VMs in each group of the network, even those without PeasantVMs. Although this design would allow for more accurate measurements between groups at any given time, it would also mean significant extra cost. For example, if a user had only $n = 5$ jobs ($0.50/minute), they would need to pay an extra $2.40/minute. RAMPS instead spawns SerfVMs as needed to explore new areas of the network, reducing measurement cost.

4 Analysis

In this section, we analyze the system's performance via complexity analysis. Then we navigate through a few test cases to expose the procedure of the system where we illustrate user experience. Finally, we conclude by discussing limitations of the system.

4.1 Performance

We note that our system does not achieve the minimum space or minimum time possible but rather attempts to balance the two realizing trade-offs that in turn provide the best space-time solution.

4.1.1 Space Complexity

In terms of space, the system employs extra space in order to explore the data center more quickly as well as isolate the decision making component of the system. The CzarVM is largely responsible for performing the optimizations on the system and does not perform the actual transfers presented in the B matrix. This allows us to isolate the decision making process and efficiently transfer the role of the CzarVM as needed. Thus we have an extra VM for this purpose.

In addition, we employ SerfVMs when exploring new spaces so that the system has accurate measurements of these regions. Since in our model we consider migration costs to be neglible, we can transfer the SerfVMs in a rapid manner. Thus SerfVM deployment is limited such that it is a small fraction of n, the total number of VMs for the computation. We set an upper bound on the number of SerfVMs at a given time to 10. The steady state value is closer to the range of 0-2 depending on the fluctations of the network.

Hence, in terms of VMs we use at most $n + 11$ at a given time, while the average value is closer to the range $[n + 1, n + 3]$.

4.1.2 Time Complexity

In terms of time, the system largely scales with the magnitude and density of the B matrix. A larger valued B matrix indicates we are transferring larger quanities of data, while a more dense B matrix suggests we are transferring among a larger number of parties. Matrix density is the fraction of non-zero elements in the matrix.

Our system in the steady state is estimated to be capable of sending in average-case $r = 1$ GB/minute (accounting for data production, low throughputs and congestion) through any given TCP connection. Hence, if we let M be the largest value in the B matrix (most bytes needing to be transferred between a pair of VMs) and ϵ measure the density of the matrix such that $\epsilon \in [0, 1]$, $M \cdot n$ serves as an upper bound on the total amount of data that needs to be transferred. A crude time estimate is given as,

$$T \approx \frac{\epsilon \cdot n \cdot M}{n \cdot r} \approx \frac{\epsilon \cdot n \cdot M}{n \cdot 1 \text{ GB} \cdot \text{minute}^{-1}} \approx \frac{\epsilon \cdot M}{1 \text{ GB} \cdot \text{ minute}^{-1}} \tag{4.1}$$

where we have taken the time with just the CzarVM during initialization to be neglible. To see that this is a reasonable figure consider $\epsilon = 0.5, n = 100, M = 10$ GB. Then we estimate that the computation takes 5 minutes.

$$T \approx \frac{0.5 \cdot 10 \text{ GB}}{1 \text{ GB} \cdot \text{ minute}^{-1}} \approx 5 \text{ minutes}$$

4.1.3 Measurement Overhead

In areas actively being used by VMs, the system will detect network changes within 1s via probing and measurements done on individual VMs. The CzarVM checks these scores every second, so the overall system learns of and begins adapting to changes in 1s. The process can take longer depending on the performance of connections to the CzarVM, but since the CzarVMs slot is auctioned off periodically, this will not have a significant impact. In unexplored areas of the network, it may take up to ~ 5 seconds for the CzarVM to identify areas with several openings and use SerfVM metrics to evaluate performance in those groups. However, about half of the worst performing VMs will be moved at once, allowing the system to adapt quickly once these areas are identified.

Each measurement we make is on the order of tens to hundreds of bytes at most. While we do transfer these metrics every second, we expect that for a given minute of computation we have at most 10 kB for a path. Comparing this to the amount of data transferred per path for actual computation, which is somewhere from 100 MB to 10GB per minute, we expect that the measurement overhead is at least 4 orders of magnitude less than the computation network bytes. Hence, it is safe to assume in this theoretical framework that the measurement overhead impacts the computation transfer network very minimally. Nevertheless, our system has built in adaptation to scale back active probing as needed.

4.2 User Experience

Definitions: The *traffic matrix* represents the amount of data transfer between the n PeasantVMs at a given moment in time. *Traffic graphs* are the graphs created from the traffic matrices, where the nodes are VMs and the edge weights are the corresponding values in the traffic matrix.

RAMPS performs particularly well for traffic matrices corresponding to disjoint clusters of VMs, with high data transfers within those clusters. An example of such a traffic matrix and its corresponding traffic graph are shown in Figure 4a. The system will initially place VMs into groups with the most occupancy, exploring areas of the network with the most open slots. The initial placement may break these cluster boundaries. However, the scoring metric will immediately identify VMs that have been separated from their clusters; the single transfer algorithm will move them to be in the same group as the other VMs in the cluster. The jobs within these clusters can thus transfer data amongst themselves very quickly, without being affected by performance in other clusters. This is particularly useful for large applications that require minutes of computation time, as the system stabilizes within seconds. Figure 4b shows a similar situation with a sparse traffic matrix, in which some VMs are not actively sending data. In this case, clusters of VMs transferring data will be grouped together, potentially (but not necessarily) isolated from those that do not.

Dense traffic matrices (Figure 4c) and upper-triangular matrices (Figure 4d) correspond to VM communication that cannot be easily partitioned into distinct clusters. In this case, if there is a large number of jobs, VM communication will necessarily cross group boundaries, potentially leading to higher cost. However, realistically, changing data transfer rates will lead to the formation of temporary clusters of communication amongst VMs. RAMPS can adapt to these formations via single/bulk transfer in seconds, thus reducing the effects of large, cross-group communication.

RAMPS's placement/movement algorithm attempts to minimize cost in the presence of other users with various configurations. When a single user fully-occupies multiple groups, RAMPS places VMs into other groups with the most openings and uses bulk transfer to utilize the new space once it opens up. In the case where each group in the data center is partially-occupied, but still has a significant number of open slots (about half), RAMPS still groups together VMs as much as possible. Single transfer allows the system to move towards the optimal clustering of jobs, even with substantial group occupancy. The worst case is when

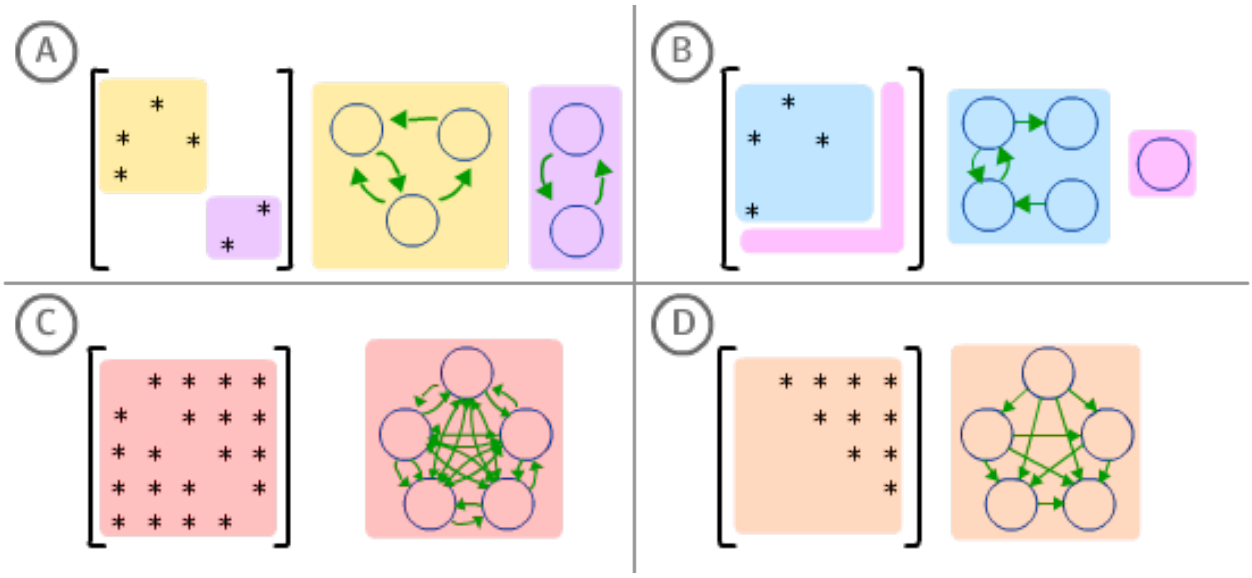

Figure 4: Panels depict various *traffic matrices* and corresponding *traffic graphs*. (A) Through row operations, the traffic matrix consists of submatrices which are topologically equivalent to independent groups. (B) A sparse matrix with a free variable representing a free allocation for a VM. (C) A dense matrix representing bidirectional computation in a group. (D) An upper triangular matrix representing unidirectional computation in a group.

there are few, highly-scattered openings in the data center. Then, RAMPS relies on single transfer to adapt to fluctuations in network conditions and find the best current configuration of VMs. These cases are further discussed in Section 4.3.

4.3 Use Cases

In this section we examine several use cases of the system. Figure 5 presents theoretical predictions for the performance expected.

4.3.1 3/4 Occupied Data Center

We first examine the case where every machine in the data center already hosts 3 VMs and we must launch a computation with 100 VMs. Our system will be at a local optimal while a global optimal might not be obtained, as our system design does not apply to other users.

In the initialization, the CzarVM ensures each group is filled identically by arbitrarily assigning machines to groups. As previously shown, our steady state migration technique smooths the system towards the minimal time, and thereby the least cost. Though it is limited by capacity and location of available VMs, our system avoids this issue by searching all available space to determine the placement for the fastest finishing computation. Despite being limited with the capacity and location of VMs available, our system avoids this issue by incrementally searching the entire space of availability to determine which placement is most likely to finish computation faster overall.

Suggesting some numbers we might suspect that with $N = 100$ VMs if we have the largest byte transfer as $M = 1$ GB between any two VMs with $\epsilon = 0.75$ and $r = 500$ MB/minute that the computation takes ~ 1.5 minutes. Thus we expect to pay \$15 for this computation. Note that we pay slightly more than this figure when we add on CzarVM and SerfVM times; though this is generally minimal compared to the PeasantVM

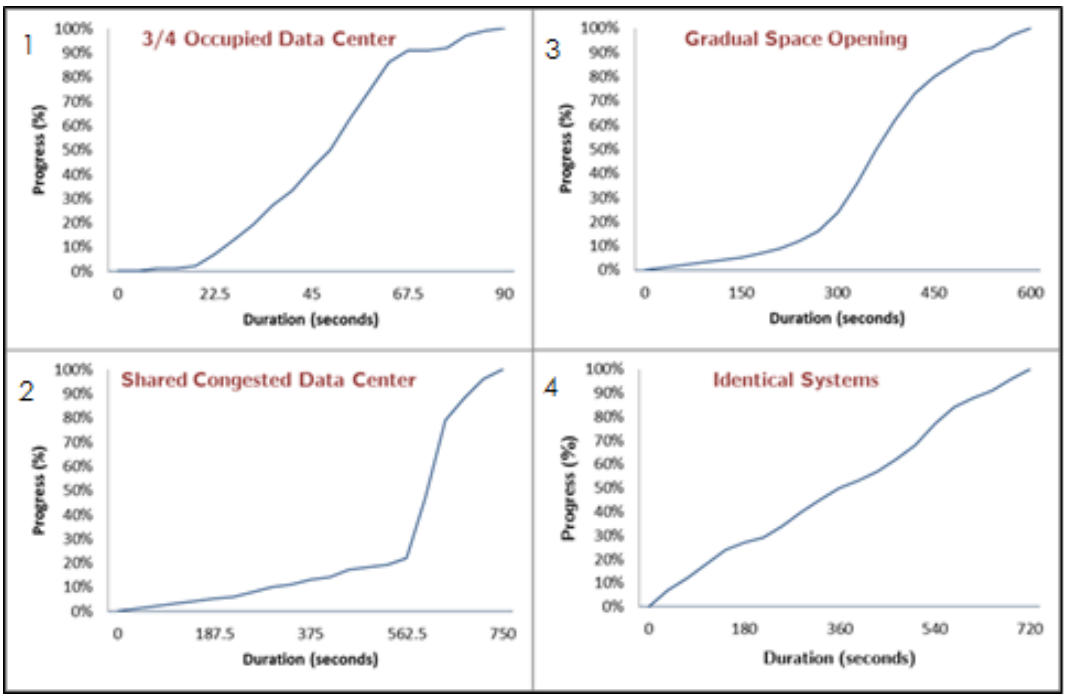

Figure 5: Graphs indicating the theoretical performance of RAMPS in the use cases. (1) Once high performing machines are located and secured the system begins performing at a steady rate. (2) The system waits on the other user to give up the high throughput group and then performs a bulk transfer. (3) The user steadily adapts to an increase in available machines with bulk transfers smoothing out. (4) Each user performs a relatively steady computation in their local region.

time as computed here.

4.3.2 Shared Congested Data Center

In this circumstance, we have deployed 10 VMs in the first 6 groups of the data center which are heavily congested. Another user has a large application that completes in groups 19-24 which has high throughput and we would like to once again optimize our placement and movement of VMs.

We are already underway in the computation in this use case. Since we have DukeVMs in the groups we are computing on, we have information about the slow throughput. The CzarVM is consistently monitoring other groups by dispatching SerfVMs. The CzarVM is notified and begins a bulk transfer to further explore the full space and see if speeding up overall computation is possible.

The bulk transfer utilizes the persistent score ranking and moves half of the worst performing VMs and shifts them to the newly-available group. This is not a perfect solution, so our steady state scoring will smooth out any exacerbated situations.

Suggesting some numbers, we might suspect that with 10 VMs if we have the largest byte transfer as 50 GB between any two VMs with $\epsilon = 0.25$ and $r = 1$ GB/minute, that the computation takes ~ 12.5 minutes. Thus we expect to pay \$12.50 for this computation.

4.3.3 Gradual Space Opening

In this circumstance, there are 100 VMs deployed across groups 1-6 with varying throughput and the rest of the network contains other users' VMs. Over time, space opens up gradually across groups rather than in one group at a time.

The CzarVM has a ready priority queue of worst performing VMs, so it detects that spots have opened up and uses the SerfVMs' metric to decide whether to move its VMs. The steady state movement protocol is used here. Once enough empty slots $N/2 = 50$ have been located, the bulk transfer technique is considered. The open slots are not restricted to a specific group, but rather distributed across groups under the same aggregate router. We bulk transfer across the groups and gradually use the steady state technique to fine tune the allocations.

Our system adapts in a slower manner when compared to a single group becoming empty. Nevertheless, we reach the optimal solution with fine-tuning such that if we have 100 VMs and a total of 20 GB to transfer with $M = 20$ GB, $r = 1$ GB/minute, $N = 100$, and $\epsilon = 0.5$ we expect ~ 10 minutes to complete as we are initially contrained to a less ideal allocation of groups and VMs and expect to pay \$100 for this computation.

4.3.4 Identical Systems

In this situation each user in the data center is using the same system as we have described in this paper. We demonstrate that each user in the system will secure its own optimal configuration.

Due to the fact that our system is largely deterministic in decision-making, each user will have the best movement or placement decision made for them. This in turn implies that each user secures a local optimal for their own computation. Furthermore, our system does not bias in favor of a single user but rather employs a greedy approach for each user and we claim that this is one of the ideal configurations.

Suggesting some numbers we might suspect that with 15 VMs if we have the largest byte transfer as 3 GB between any two VMs with $\epsilon = 1.0$ and $r = 250$ MB/minute that the computation takes ~ 12 minutes. Thus we expect to pay \$18 for this computation.

4.4 Limitations

One limitation of the system is the centralized decision making procedure by the CzarVM - a single point of failure. Although this offers advantages of making quick decisions and avoids distributed decision making, it could damage the system if the CzarVM performs poorly or loses data. Steps in fixing this issue include auctioning to transfer control when performance is poor. Also borrowing from the development of RAID, we can distribute replication on the VMs for redundancy of system data if VM failure is truly a concern [4].

Another limitation of the system is the steady state movement procedure. The system simply moves one VM at a time which would then smooth to achieve the optimal. However, delaying movement and increasing computation can lead to optimization over a *set* of VMs and may offer less cost if the `move` operations were costly (we assumed neglible cost in this model).

Lastly, many metrics that might serve of practical importance in real world data centers such as CPU utilization and queue lengths are not considered for simplicity as we have assumed that the machines perform effectively.

5 Conclusion

RAMPS minimizes VM usage cost by using a combined metric of throughput and latency to reduce overall computational time. The system balances the use of active and passive probing to detect changes in network conditions. It also combines single and bulk transfer to adapt to faster connections, increasing throughput and lowering cost. The analysis shows that while the design does not achieve the minimum space or time usage possible, it assesses both trade-offs to achieve a best combined space-time solution, with limited overhead for measurements. Future work on the system involves replicating the CzarVM to eliminate the single point of failure, as well as handling machine failures.

6 Acknowledgements

The authors would like to thank Dr. Karen Sollins for her feedback on the design, Brian Rosario for his assistance in gleaning important parts of the system, and Dr. Janis Melvold for helping develop the presentation.

7 References

[1] Corbett, James C and Dean, Jeffrey and Epstein, Michael and Fikes, Andrew and Frost, Christopher and Furman, JJ and Ghemawat, Sanjay and Gubarev, Andrey and Heiser, Christopher and Hochschild, Peter and others, "Spanner: Google's globally distributed database." *ACM Transactions on Computer Systems (TOCS)* 31, no 3 (2013): 8.

[2] Alizadeh, Mohammad, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. "Data Center TCP (DCTCP)." *ACM SIGCOMM Computer Communication Review* 41, No. 4 (2011): 63-74.

[3] Saltzer, J. H., and Frans Kaashoek M., "Principles of Computer System Design: An Introduction", 1st ed., Burlington, MA: Morgan Kaufmann, 2009.

[4] Patterson, David A., Garth Gibson, and Randy H. Katz. "A case for redundant arrays of inexpensive disks (RAID)". Vol. 17, no. 3. *ACM*, 1988.

Wordcount: 4981