GreedyCycle: A Cycling Algorithm for Optimizing Cost Right Now

Miren Bamforth (miren@mit.edu)
Jennifer Hu (hujj@mit.edu)
Victoria Li (vicli@mit.edu)
Day/Trice (R11 & R12)
6.033 Design Report 2
May 9th, 2014

1 Overview

Users of large, shared data centers desire to spend the smallest possible amount of money when running their programs. Without a measurement algorithm, the user's program will not be able to respond to fluctuations in network congestion or machine availability. Furthermore, without a placement algorithm, the user's program will have no way to harness network measurements. We propose a solution to the data center's inherent measurement and placement challenges in the form of GreedyCycle.

GreedyCycle maximizes the probability that all of a job's information will be initialized to adjacent machines or groups of machines. GreedyCycle uses throughput measurement and machine placement data to move either a few or all of the user's virtual machines in one algorithm cycle. Applying GreedyCycle's algorithms to a job will consistently decrease the overall runtime and save the user money. While not being the mathematically optimal long-term solution, GreedyCycle's greedy solution will always optimize a job's placement as much as possible without incurring unacceptable overhead costs like a mathematically optimal solution would.

2 Design Description

GreedyCycle's design consists of one measurement and three placement algorithms. The measurement algorithm explains which measurements we consider necessary for determining which VMs to move. The initialization algorithm determines initial placement of VMs. The primary placement algorithm moves one VM at a time to improve the performance of the worst links. The secondary placement algorithm detects large changes in the network and sometimes moves all of the VMs at once. Throughout this section we state previous iterations of our algorithm design and why the final design was chosen.

We will define terms, state assumptions, and describe the system structure.

2.1 Terms

- A **virtual machine (VM)** is a computational component within the system
- A **job** is a user's program
- A **worker VM (worker)** is a VM which is running the user's job
- A **placement master (PM)** is a VM which controls placement of other VMs. There is one per job. It is not also a worker VM
- A **probe VM (probe)** is a worker VM which the PM temporarily repurposes to measure the network
- A **group master (GM)** is a special worker which controls measurement information for all VMs in its group and talks to the PM. There is one GM per group

We use the terms given by the data center's arrangement in Figure 1. The most important terms are a **group** of 48 machines with one group router and a **cluster** of 6 groups with two aggregate routers.

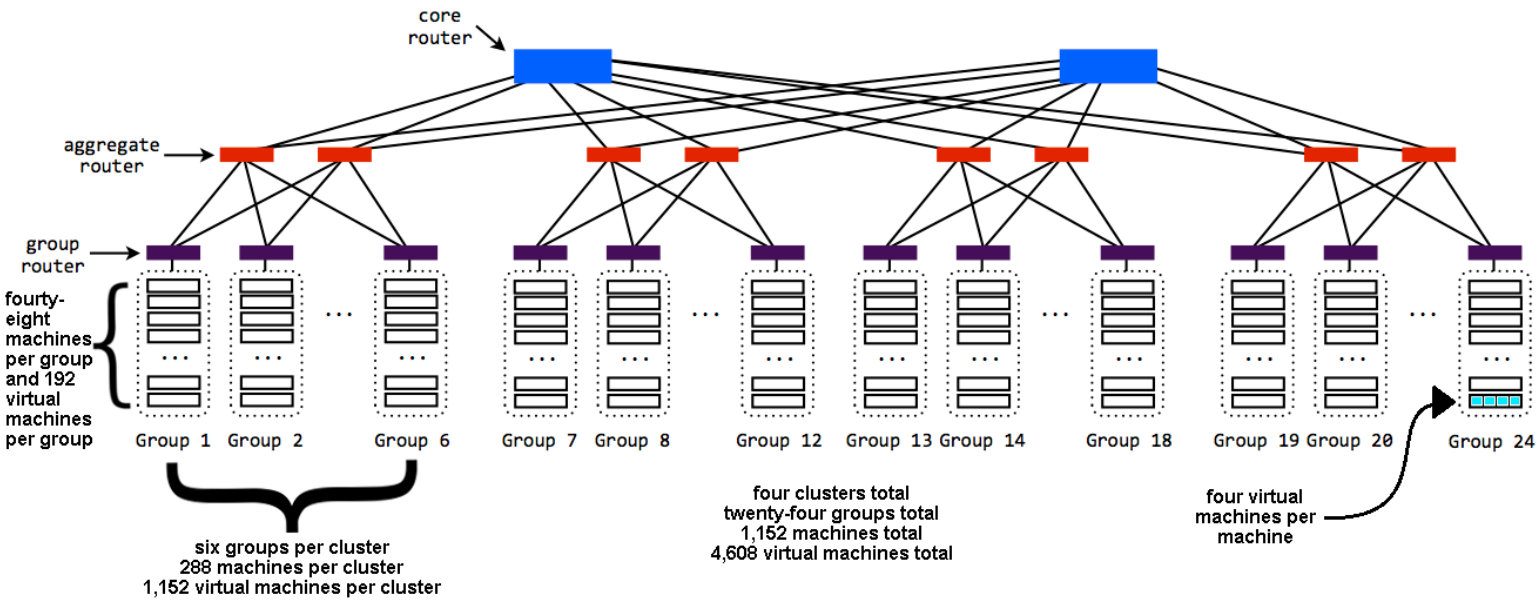

Figure 1: The preset data center structure [1]

2.2 Assumptions

To simplify the design, we make assumptions that fall into three categories: speed of the network, network conditions, and user desires.

2.2.1 Speed of the Network Components

We assume that VMs on the same machine have 'throughput' limited only by machine processing speeds. These inner 'links' are faster than network links because the data center's bottlenecks are due to "communication bandwidth not CPU or memory" [2]. We assume that machines have a connection of 100 GB/s to and from their group router [3]. Finally, we assume that the group, aggregate, and core router links are all 10 GB/s links [1].

With these assumptions, we can postulate an ideal VM placement to minimize runtime. VMs on the same machine have the fastest connection, so having as many VMs on the same machine as possible is the most desirable configuration; this arrangement would be best for a very small job with at most four VMs or for a transfer-heavy subset of four or fewer VMs from a larger job. Similarly, machines within a group have the second fastest link, so the next most desirable configuration of the system is to have as many VMs as possible within the same group. While this arrangement works for jobs or transfer-heavy job subsets of 192 VMs or less, having VMs in multiple groups is often unavoidable, especially for larger jobs.

2.2.2 Network Conditions

The greatest hindrance to GreedyCycle's algorithms is that the other users act unpredictably. As a result, GreedyCycle - or any network system not privileged to others' jobs - can only optimize with the past and present in mind; there is no way to predict and optimize for future conditions. As explained in §2.4 and §2.5, GreedyCycle chooses to move VMs to maximize the amount of time that can be saved *right now* without considering that a link might improve on its own as network conditions change. GreedyCycle uses greedy algorithms because it is only concerned with the present.

GreedyCycle's algorithms run once every *algorithm cycle*. We determine a reasonable length for an algorithm cycle to be two seconds for the primary algorithm and ten seconds for the secondary algorithm. GreedyCycle could respond to network changes more quickly, but we assert in §3 that using little bandwidth like GreedyCycle does is worth responding at these rates.

Finally, we assume that there is a negligible cost or delay for functions in the API, including the functions which move VMs [1].

2.2.3 Desires of the User

We assume that the user would like their job to start as soon as they submit it to the data center. As a result, we spend very little time doing computation before deploying all worker VMs and starting the job.

2.3 System Structure

We organize our system to work well with the data center's structure. This section explores VM layout, VM removal, and the data center API.

2.3.1 Virtual Machine Structure

The first VM placed in the data center is the **placement master (PM)**. This master processes data sent to it by the group masters and makes decisions about moving VMs. The PM stores various data as shown in *Figure 2*, and sends **probe VMs** to find large network changes.

Previously, we originally tried to avoid having any extra VMs, but the convenience of a PM overrode the difficulty of implementing a system with distributed control.

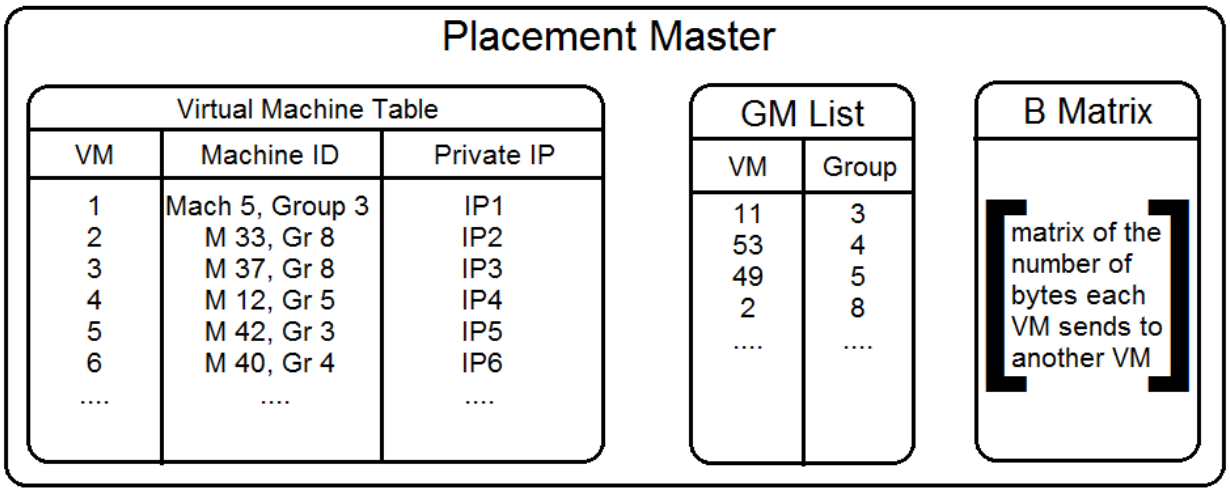

Figure 2: The PM's data structures

We designate one worker from each non empty group to be the **group master (GM)**. The first VM to be placed in a group is that group's GM. The GM collects data from each worker in its group and sends requests to the PM based on this data, as described in §2.4.

2.3.2 Removing Finished Virtual Machines

Worker VMs call *remove_finished(v)* (explained in §2.3.3) on themselves once a second since there is no cost to call methods [4]. A worker informs its GM when finished and will remove itself once the GM acknowledges its message. The GM then tells the PM to remove the finished VM from the VM table. If the worker is also the GM, it tells the PM to assign a new GM. The new GM tells the workers in its group that it is now the GM.

The PM removes itself once all of the remaining VMs are on one machine since it can no longer improve the placement.

2.3.3 GreedyCycle API

The user does not call methods in the API; the methods are called by the VMs. The following methods are provided by the data center.

- *IP_addr place(v, m):* Places VM v on machine m. Returns the private IP address of the VM if successful and null if not.
- *(machine_id, IP_addr) random_place(v):* Places VM v on a random machine. Returns the machine's ID and v's private IP. Guarantees placement unless the data center is too full, in which case it returns an error code.
- *int progress(u, v):* Returns the number of bytes that virtual machines u and v have left to transfer to each other.

- *int machine_occupancy(m):* Returns the number of VMs currently running on machine m.
- *double tcp_throughput(v):* Returns the throughput of the TCP connection from this VM to VM v. The throughput is computed over the last 100ms.

We also define a set of methods specific to our system.

- *int group_occupancy(g):* Returns the number of VMs currently running on the machines in group g by calling *machine_occupancy(m)* on each machine in the group
- *int directed_progress(u, v):* Returns the number of bytes that VM u has left to send to VM v. Note that this is not the total bytes left in both directions.
- *boolean remove_finished(u):* Removes virtual machine u if *progress(u, v)* returns 0 for all links involving u. It tells its GM that it is done and removes itself from the system when it receives an acknowledgement from the GM. Returns true if the VM is successfully removed or false if *progress(u, v)* is nonzero for any link.
- *int data_in_buffer(u, v):* Returns the number of bytes that virtual machine u has in its output buffer ready to be sent to virtual machine v. Since the VMs take time before they are ready to send data, there is a limited amount of data which is ready to be sent at a certain time; this data sits in the output buffer and can be quantized by this function.
- *boolean set_GM(v):* Sets VM v to be the GM in its group; returns true if successful.

2.4 Measurement Algorithm

This section explains what we measure, how we measure, and how the measurement algorithm works.

2.4.1 Measurement Definitions

We define our measurements below for the link from VM *u* to VM *v*.

- Desired throughput, **D**, is the throughput that VM u would like to have, defined as *data_in_buffer(u, v)*.
- Current throughput, **C,** is equal to *tcp_throughput(v)*
- **I** is *directed_progress(u, v)*, the remaining bytes to send
- **S** is the number of seconds that could be saved if the link were transferring data at the desired throughput, calculated as **S = I/C - I/D.** S is a useful metric because it quantizes how the VM placement could be improved *right now*, as desired by the user and stated in our goals for the system.
- A **clump** is a set of VMs in the same group which have links to the same VM within another group.
- A **problem VM** is the VM which all of the VMs in a clump have bad links to; moving the problem VM improves all the links of a clump

- **S_total** is the number of seconds that could be saved by moving one VM to satisfy a clump
- The **worst link** is defined as having the largest **S**

2.4.2 Worker Measurements

Each worker thinks of itself as VM *u*, and each VM that it must send data to as VM *v*. Each worker then calculates S for every link it has by using the equation and definitions above. It then sends the S for each link to its GM.

2.4.3 Group Master Measurements and Clump Calculation

Given the data from each worker, each GM finds clumps within each group. To find a clump, the GM groups together all links where the VM obtaining the transferred data is the same; this receiving VM is the **problem VM**. For example, in Figure 3, there are two clumps, the set {A} with problem VM F and the set {B, C, D, E} with problem VM G.

The GM calculates S_total for each clump by summing S for each link in the clump. In our example, S_total for the set{A} is 10 seconds. S_total for the set {B, C, D, E} is 20 seconds.

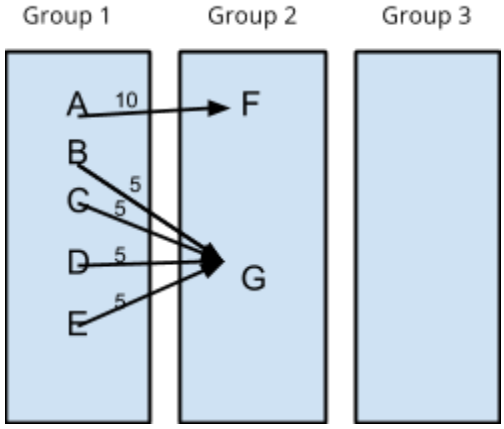

Figure 3. The numbers above each link denote that link's S

Each GM sends **clump data** to the PM as shown in Figure 4. The clump data sent corresponds to the clump with the greatest S_total. Moving the problem VM of this clump could save the most time overall, so it is the best problem VM to move. If two clumps have the same S_total, the one with the most VMs is chosen. For the example in Figure 3, Group 1's GM will send "greatest S_total is 20, problem VM is G, number of VMs is 4".

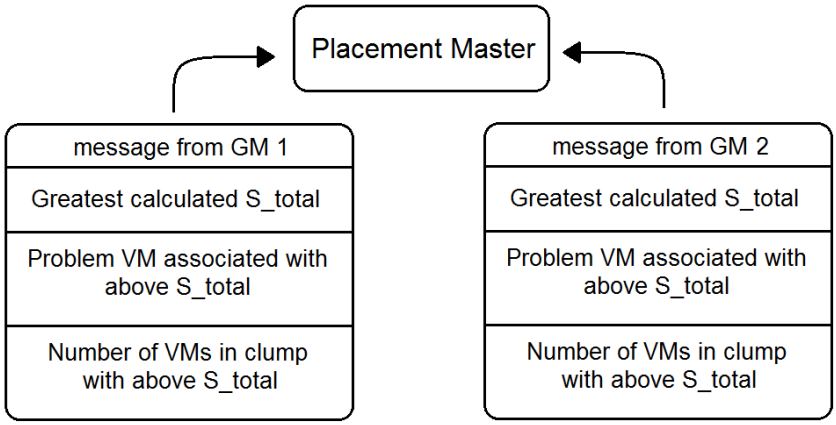

Figure 4. The structure of the clump data

Each GM in the network sends this clump data to the PM once every two seconds; we justify this algorithm cycle time in §3.1. The PM uses this data in the primary placement algorithm as described in §2.5.2.

Measuring the network in terms of time that could be saved if we move a specific VM *right now* is in line with our goals and assumptions stated previously.

2.5 Placement Algorithms

We place the VMs with three algorithms: the initialization algorithm for initial placement, the primary algorithm for improving the worst links, and the secondary algorithm for moving the entire job to a less congested part of the network.

2.5.1 Initialization Algorithm

GreedyCycle maximizes the probability that all VMs are initialized to adjacent machines or groups of machines because we assume that adjacent placement is desirable. The initial placement has two steps: placing the first VM and placing the rest of the VMs.

The first VM is placed by *random_place(v)*. This VM is the placement master. The PM then does the following steps:

1. It calls *group_occupancy(g)* on each group, including its own. This takes 1 RTT to compute because *group_occupancy(g)* must receive an answer from each machine.
2. It moves itself to the emptiest group using *place(m,v)* until it is successfully placed on a machine with space.

As opposed to an alternate solution of placing the PM once, we posit that randomly placing the PM, taking measurements, and carefully moving the PM allows the PM to put itself in an ideal location. The rest of the VMs are placed as follows:

1. The PM attempts to place the VMs within the same group as the PM.
2. Once a group is full, VMs are placed on machines in a different group within the same cluster.
3. Once a cluster is full, VMs will be placed in a group on another cluster, and likewise until all VMs are placed or there is no space left.

This VM placement is based on the assumption that connections between same-cluster groups will probably be better than connections between groups in different clusters. While this is not always true, we assume it is most likely true for the sake of initial placement. The primary placement algorithm cleans up any resulting bad links. This initialization is better than randomly placing all VMs because all VMs are placed in as few groups as possible which is desirable as stated in §2.2.1.

The first VM (other than the PM) placed in each group is the group master. A group without workers has no GM.

2.5.2 Primary Placement Algorithm

The PM moves individual VMs to improve the worst links after the initial placement. As mentioned in §2.4.3, each GM sends clump data to the PM. The PM uses the clump data to determine which VMs should move.

The PM finds the clump with the maximum S_total from all the clump data and moves the associated problem VM to the group where its corresponding clump is. Then, making sure never to move a problem VM twice in an algorithm cycle, the PM moves the problem VM associated with the next biggest S_total to its corresponding group, and so on. The PM informs the relevant GMs that they have lost or gained a VM; the PM also updates its own data structures described in 2.3.1, including the GM list if a GM is moved or created.

By abstracting calculations to the GM, we minimize the amount of data sent to the PM and the computation time needed for the PM. The GMs save bandwidth by not sending the S of each link to the PM, and the PM receives at most 24 clump data values per algorithm cycle, allowing for efficient computation in the PM, as opposed to receiving data from all of the potentially hundreds of VMs in the job.

For alternate solutions, we considered various ways to improve one bad link at a time, but these approaches did not fix the issue of a poorly placed VM causing most of the lost time. Our primary placement algorithm is able to fix one bad link or many at once to combat the negative effects of

one problem VM. We would need a shorter algorithm cycle and therefore more overhead if we fixed only one link at a time.

2.5.3 Secondary Placement Algorithm

Every 10 seconds, GreedyCycle probes a cluster to detect large changes in network conditions. Probing is the only way that GreedyCycle can discover large changes in the network. We use worker VMs as probe VMs to save space and money; creating 'ghost VMs' whose sole purpose is probing uses extra space, incurs unnecessary cost, and is an unacceptable alternate solution.

We explain the time each step takes in terms of the round trip time (RTT) of packets from the PM to other VMs as justified in section 3.1:

1. The placement master designates the cluster that has been probed least recently as the next cluster to probe; it does not choose the emptiest cluster because that cluster is not guaranteed to have the best conditions solely because of its occupancy. The PM checks if the cluster has enough space to contain all of the job's remaining VMs by calling *group_occupancy(g)* on each group in the cluster. If not, the PM checks the occupancy of the next cluster probed least recently, and so on. If no clusters have enough space, the algorithm ends. This takes between 1 and 4 RTTs because *group_occupancy(g)* takes one RTT and may be called up to four times.

2. Two workers are chosen to become probe VMs temporarily in a way which interrupts the job as little as possible. If there is a solo VM or a pair of VMs which are the only VMs in that group, these VMs are chosen to be probe VMs. Otherwise, a pair of non-master VMs from the group with the fewest workers is chosen; we choose the pair with the most data to send to each other out of the VMs in that group. Therefore, if these probes get stuck elsewhere in the system after completing the dynamic placement algorithm, they can continue to send their large amounts of data to each other until the primary placement algorithm moves them back to the other VMs. Choosing the probe VMs is essentially instantaneous because the PM refers to its VM table and sends no messages.

3.
    a. The probe VMs are deployed to the cluster in question with *place(v, m)*. They use active probing by sending packets as quickly as TCP will allow for 300ms. 300ms should be enough time for an uncongested network to attain an accurate throughput; some links might not achieve good throughput in 300ms, but these links are therefore too slow for our desires anyway. Then, they call *tcp_throughput(v)*, record the results, and move themselves to a different pair of groups to measure again. They do this for all fifteen unique group pairs in the cluster. When finished, they send the measurement data to the PM. This step takes 4.5 seconds (15 measurements * 300ms/measurement) plus one RTT to send the data to the PM.

b. Meanwhile, the PM asks the GMs to compute an average *tcp_throughput(v)* for each link of each VM in their groups. This takes approximately one RTT.

4. The PM uses the measurement and occupancy data to determine if moving to this cluster would be better than the current situation. Primarily, if the VMs can be moved into a smaller total number of groups than they are in now, the PM moves everything, as having as many VMs as possible within a group is a desirable configuration. Otherwise, the PM calculates the average throughput from the probe measurements. If this average throughput is greater than the average throughput measured by the GMs in part 3b, the PM moves everything. This step is essentially instantaneous because it involves no packet sending.

5.
    a. If desired, the PM moves every VM to the new cluster and assigns new GMs. The PM moves the VMs to the emptiest group in the new cluster; if the new group becomes full, the remaining VMs are placed in the next emptiest group. If another job steals space and there are remaining VMs, they are left in place for the primary placement algorithm to clean up. The PM moves itself last. This step is instantaneous since moving VMs has no cost.
    b. If the PM decides not to move everything, the two probes are returned to their previous groups; if their groups are full, they are placed in the same cluster. If the cluster is full, they are left where they are to continue their regular calculations; the individual placement algorithm will move them. This step is instantaneous because place(m, v) is negligible[1].

The total time to completion is **2 to 5 RTT plus 4.5s**.

Since moving VMs has negligible cost, there is no downside to shifting the whole system at once. The small risk of another job stealing space while the PM is processing is worth the potential lower runtime. The primary placement algorithm cleans up any inefficiencies caused by the secondary placement algorithm.

Ten seconds is an algorithm cycle length which creates a balance between disrupting the job too often by sending many probes and finding empty sections of the network before other jobs.

3 Analysis

3.1 Time Efficiency

The core part of our time analysis is the round trip time (RTT). We utilized the Network Utility tool to ping various servers and calculate a reasonable average RTT to use for our calculations [5]. We pinged servers at MIT and at Akamai; presumably, the data center links have faster

connections and less distance to travel than the links that we pinged, so the RTTs from these calculations give an upper bound to GreedyCycle's RTT.

20 ping tests with packet size of 56 bytes were conducted for the following addresses: csail.mit.edu and akamai.com over the course of 3 days.

Server	**Minimum (ms)**	**Average (ms)**	**Maximum (ms)**
csail.mit.edu	1.621	2.632	37.000
akamai.com	19.773	24.370	102.519

Figure 5: Ping data

In order to account for larger packet sizes and possible faulty links, we use the maximum RTT time (102 ms) as the expected RTT and add an additional buffer of 400ms for the worst case. The worst case RTT is **500ms** and the expected RTT is **100ms** for links with multiple hops such as links between the PM and the GMs; the RTT is smaller within groups, but we will use these numbers anyway to account for the worst case RTTs.

GreedyCycle's time efficiency depends on the efficiency of the algorithms and the overall runtime and cost. Figure 6 shows numerical average and worst cases times for each step.

- **Initialization:** The PM waits for one RTT to receive the responses from each group before moving itself and placing the other VMs; this step is fast because only one measurement is taken.
- **Measurement:** When workers call *tcp_throughput(v)* on themselves, it is instantaneous as stated in our assumptions. They send data to the GM which takes one RTT. The GMs calculate the clump data; we assume that this is also negligible relative to an RTT.
- **Primary Placement Algorithm:** The GMs send the clump data which takes one RTT to get to the PM. When the PM receives the clump data, it does calculations which are also negligible in time compared to one RTT. Moving the problem VMs also takes negligible time [1]. The entire process from measuring to moving problem VM takes two RTTs; even in worst case network conditions, the algorithm will complete within one algorithm cycle because 2*500ms is less than the two second algorithm cycle time.
- **Secondary Placement Algorithm:** The secondary placement algorithm finds big changes in the network within ten seconds at best, within twenty seconds on average, and within forty seconds at worst because it probes a different cluster once every algorithm cycle of ten seconds. The time to completion is 2 to 5 RTT plus 4.5 seconds as stated earlier. Even in the worst case, it takes 500ms*5 plus 4.5 seconds which equals seven seconds. The algorithm time is ten seconds to allow the algorithm to finish before a new round it started. In terms of cost, repurposing workers as probe VMs reduces the cost of the secondary placement algorithm. A probe VM is used for 4.5

seconds, so starting and ending a new VM would incur one minute of cost for 4.5 seconds of work. Moving a worker does incur some cost because many of its links may decrease in quality, but the primary placement algorithm has over 27 algorithm cycles of 2 seconds each before the cost of repurposing worker VMs becomes worse than creating two new probe VMs.

Algorithm Step	Initialization Algorithm	Measure-ment Alg.	Primary Placement	Secondary Placement
Time in terms of RTT	1 RTT	1 RTT	1 RTT	2 to 5 RTT + 4.5s
Average Case	100ms	100ms	100ms	4.7s to 5s
Worst Case	500ms	500ms	500ms	5.5s to 7s

Figure 6: the average and worst case times for GreedyCycle's algorithms

Given enough time, GreedyCycle collects all VMs into a small number of groups, lowering the cost overall because the VMs will have the fastest possible links; unless the data center happens to randomly place all VMs together neatly in an uncongested network, GreedyCycle will without fail arrange the VMs in a better manner, improving the runtime. The user does not see GreedyCycle working behind the scenes, but the shorter runtime and therefore lower cost is certainly observed by the user. Since we do not create new VMs for probing, the user has no surprise costs associated with creating VMs for a few seconds before discarding them again. Theoretically, GreedyCycle improves the user experience, particularly for certain use cases as described in §3.3.

3.2 Space Efficiency

In terms of space efficiency within a VM, the extra measurements take up a negligible amount of the processing power of a regular worker based on our assumptions about the API. GMs and probes use a larger but still negligible amount of processing power since the algorithms only run once every two or ten seconds. The PM uses all of its processing power for updating data and making decisions since it is not also a worker. Due to the mostly negligible space usage in the VMs, the average per path bandwidth is the primary space efficiency concern.

All extra packets are smaller than 1 KB based on a .txt file containing similar amounts of information. The ‘optimal’ solution requires all of the link data being in one place so that mathematical optimization can take place.

As shown in Figure 7, GreedyCycle incurs more bandwidth compared to taking no measurements; however, GreedyCycle always uses less bandwidth than the ‘optimal’ solution which would require analyzing all data in one place at one time and therefore using a lot of bandwidth. Since GMs send information independent of job size and probing does not happen continuously, bandwidth is saved.

Link type	Worker & GM	GM & PM	PM & Probes
Average Case	Link data every 2s: 1 KB/2s * n links/ worker = n/2 KB/s	Clump data every 2s: 1 to 24 clumps /2s = .5 to 12 KB/s	15 data points per cycle = 15KB/10s = 1.5KB/s
Without GreedyCycle	0 GB/s	0 GB/s	0 GB/s
'Optimal' Solution	Link data sent per sec: 1KB/s*n links/ 1 worker = n KB/s	PM gets all data: w workers * n KB/s = w*n KB/s	Probing always: 300ms to probe, so 1KB/.3s = 3.3 KB/s

Figure 7: the per path average extra bandwidth for each type of link

Because we do not create new VMs for probing, no extra VM space is used, so it is less likely that the users will have any jobs rejected from the data center due to lack of space.

3.3 Use Cases

GreedyCycle works for a variety of use cases.

- **All machines have three VMs at initialization time:** GreedyCycle initializes as many VMs as possible into each cluster until that cluster is full, so GreedyCycle will take the fourth slot on each machine for an entire cluster which is the best configuration that can be achieved in this use case. However, since there is little empty space, the individual placement algorithm will not be able to satisfy clumps; until some VMs finish or move, GreedyCycle does not work well. If the job is small enough, the dynamic algorithm will still function. The job will continue like this until the network changes, at which point the individual placement function will move problem VMs.

- **Small job running in a congested cluster; a large job finishes elsewhere:** GreedyCycle’s dynamic algorithm works well here; it finds the empty cluster within ten seconds in the best case, forty seconds in the worst case, and twenty seconds in the

average case. There is a tradeoff of sometimes missing an opportunity to move the whole job, due to infrequent probing, but this is worth it as we are not continually interrupting the job's progress.

- **All jobs use GreedyCycle:** Since GreedyCycle incurs one extra VM per job, the extra overhead is inversely proportional to the size of the jobs. If each job has five VMs, overhead dominates 20% of the network; if each job has 100 or more VMs, overhead is less than 1%. If the data center is full of small jobs, new jobs will be rejected because of the extra space taken up, and the user will be unhappy. However, we believe that one extra VM per job is a small, necessary amount of overhead, and we accept the worst case scenario.

- **Very large job:** The dynamic placement algorithm will not run if a job is larger than 1,152 VMs or if a job is smaller than 1,152 VMs but still too large to fit fully into a group given the placement of other users' VMs. We accept this drawback because all jobs will eventually be small enough to fit into one cluster.

- **Medium job:** GreedyCycle works best for a job which is small enough for the dynamic placement algorithm to run but large enough such that the placement VM is not a large percentage of the total VMs; the preferred starting size is between 50 VMs (2% overhead) and 576 VMs (half the size of a cluster).

4 Conclusion

GreedyCycle measures the network and moves VMs to respond to present network conditions, saving the most time *right now*. Keeping efficiency in mind, GreedyCycle attempts to create as few extra VMs and as little extra bandwidth usage as possible. GreedyCycle reduces cost with simple changes to alternative algorithms such as repurposing workers instead of creating new probe VMs. Data center jobs execute more quickly when using GreedyCycle's greedy algorithms, finishing early and leaving extra money in the user's pocket.

5 References and Acknowledgements

[1] 6.033 Spring 2014 Design Project 2 Assignment: Running Jobs in a Data Center Network. http://web.mit.edu/6.033/www/assignments/dp2.html
[2] Piazza post on DP2 bottlenecks answered by Dina Katabi on 4/27/14, accessed 4/30/14. https://piazza.com/class/hp4dt9k35qp33s?cid=441
[3] Piazza post on DP2 answered by Dina Katabi on 4/28/14, accessed 4/30/14. https://piazza.com/class/hp4dt9k35qp33s?cid=438
[4] Piazza post on the validity of removing VMs once they are finished answered by Tiffany Yuhan Chen on 4/30/14, accessed 5/7/14. https://piazza.com/class/hp4dt9k35qp33s?cid=467

[5] Mac network utility tool. http://support.apple.com/kb/ht5897

Word count: 4998/5000 not including the references section, cover page, or headers