University of Waterloo

Dynamic Workload Management in Heterogeneous Cloud Computing Environments

Qi Zhang and Raouf Boutaba

University of Waterloo

IEEE/IFIP Network Operations and Management Symposium

Krakow, Poland

May 7, 2014

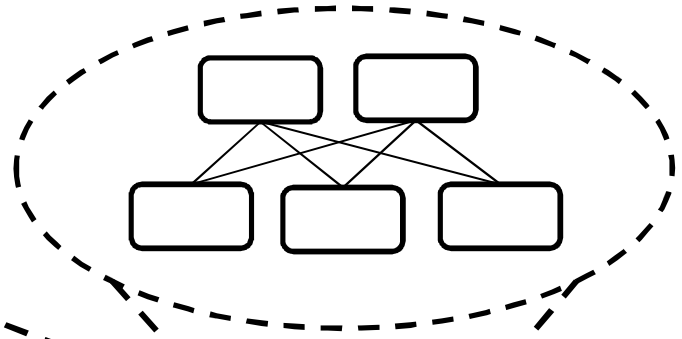
Outline

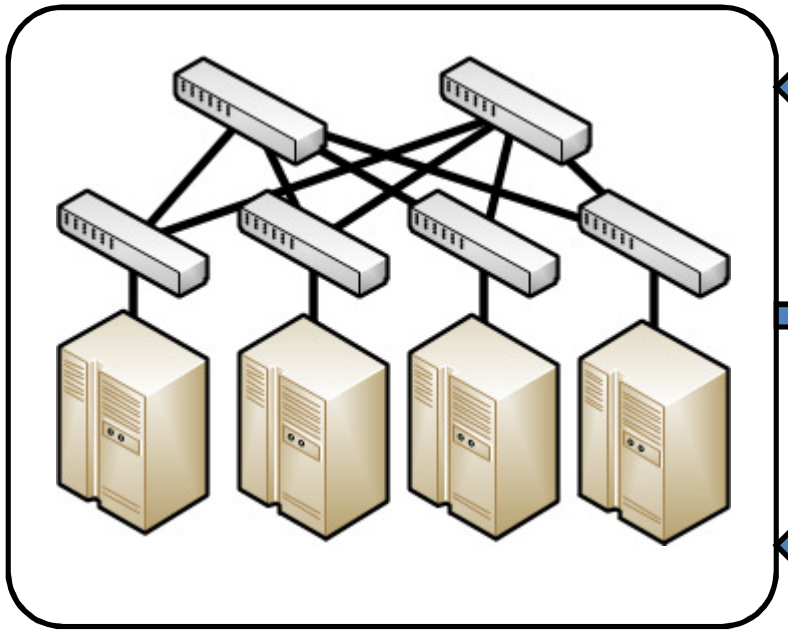
- **Introduction**
- **Cloud Workload Management**
- **Research Contributions**
- **Conclusion**

Introduction

- **Cloud computing** is a model that advocates hosting online services in data centers

Workload Management in Cloud Data Centers





Scheduler

Resource Monitor

Mgmt. Software

App. Requests

Service Provider

Cloud Provider

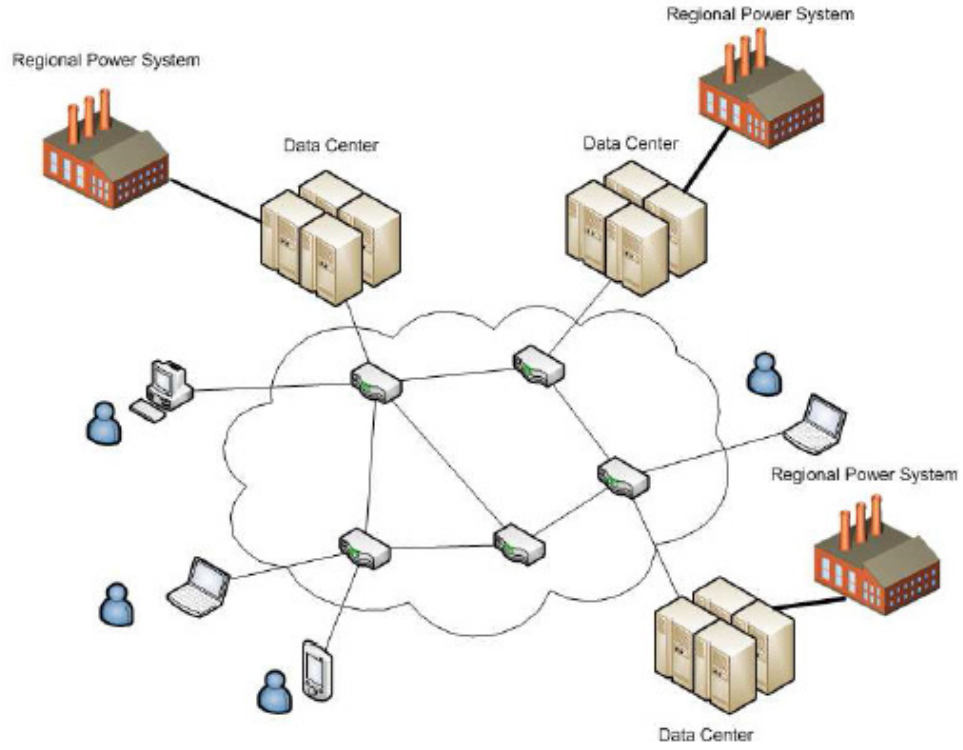
Workload Management

- Cloud workload management is difficult!
- ***Service Provider Challenge***
  - ***Dynamically*** provisioning sufficient server capacity to satisfy service level objectives (SLO), while minimizing operational cost
- ***Cloud Provider Challenges***
  - Performing resource management with consideration to ***heterogeneous*** machine and workload characteristics

Thesis Contributions

- **Service Placement in Geo-Distributed Clouds**
- **Heterogeneity-Aware Dynamic Capacity Provisioning**
- **Fine-Grained Resource-Aware Scheduling for MapReduce**

Service Placement in Geo-Distributed Clouds



- Dynamic Service Placement Problem (DSPP):
  - Where should the service be placed to reduce resource cost while satisfying service level objectives (SLOs)?

Service Placement in Geo-Distributed Clouds

- Design Challenges
  - Service demand is dynamic and originates from multiple locations
  - Electricity prices are different from location to location and can fluctuate over time
  - There is a cost associated with reconfiguration
    - Setting up the server (e.g., VM image distribution)
    - Tearing down the server (e.g., data / state transfer)
- Limitations of existing work:
  - Early studies focus on static scenarios
  - Ignoring electricity cost and reconfiguration cost

DSPP Model

Resource cost Reconfiguration Cost Performance cost Capacity penalty

$$\min_{\{\mathbf{u}_0,..,\mathbf{u}_{K-1}\}} \quad J = \sum_{k=0}^{K} \mathbf{p}_k^\top \mathbf{x}_k + \mathbf{R}^\top g_k(\mathbf{B}\mathbf{u}_k) \quad + P_k(\mathbf{a}_k\mathbf{x}_k - \mathbf{D}_k) + \pi(\mathbf{s}^\top \mathbf{x}_k)$$

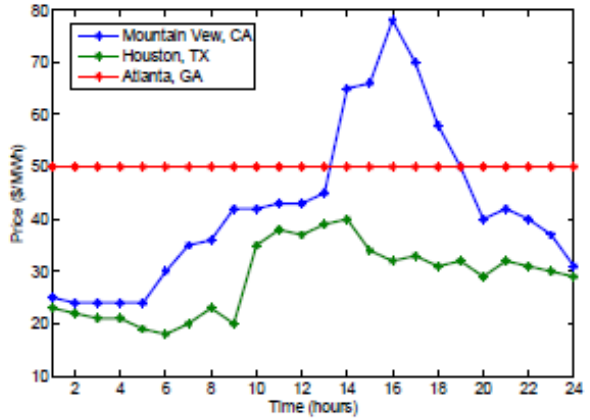
$$\text{s.t.} \quad \mathbf{x}_{k+1} = \mathbf{x}_k + \mathbf{u}_k \qquad \forall k \in \mathcal{K},$$

$$\mathbf{x}_k \in \mathbb{R}_+^{LV} \qquad \forall k \in \mathcal{K}$$

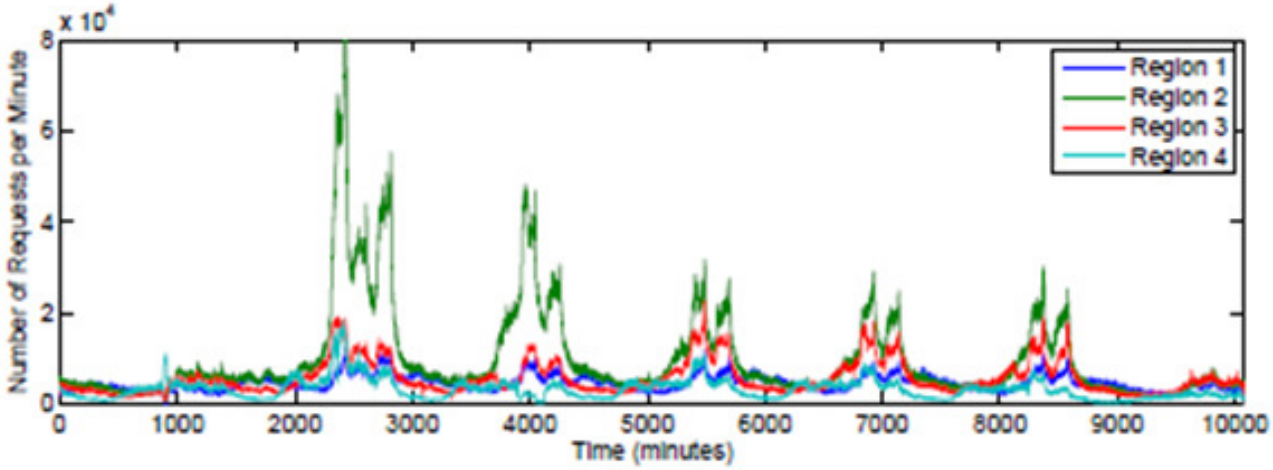
- The offline can be formulated as a discrete-time optimal control problem
- **Our solution**: online algorithm on Model Predictive Control (MPC) framework
  - Predict future demand over next K periods
  - Solve DSPP over the next K periods
  - Carry out the solution move for the next period

Results

- Rocketfuel topology
- 3 data centers

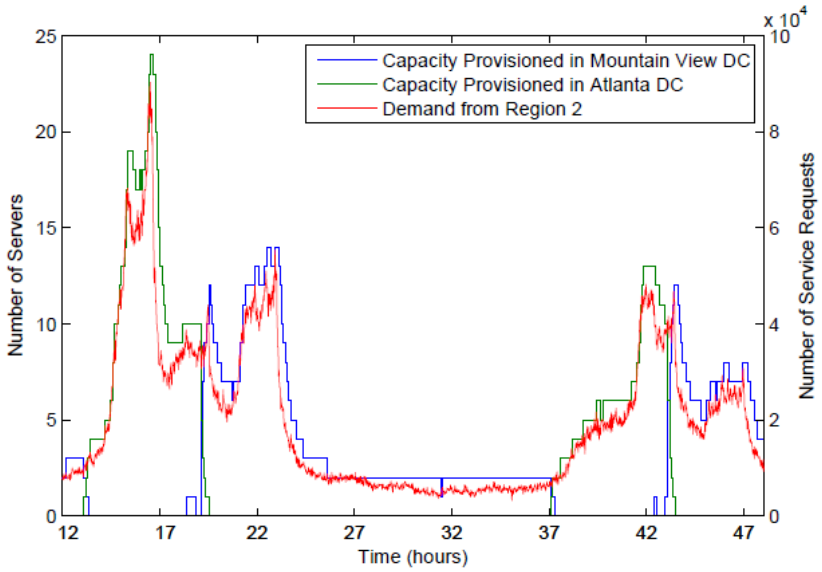


Electricity Price

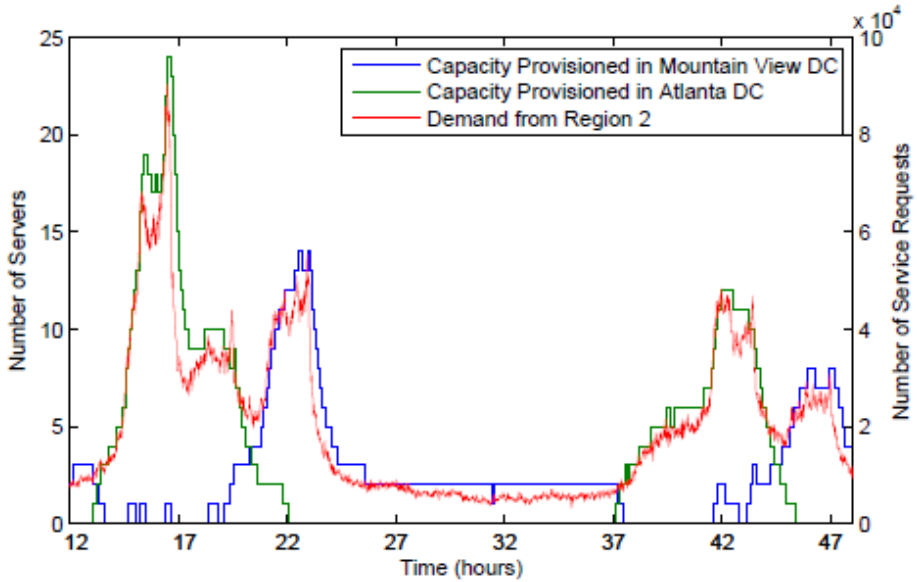


Web workload

Results



Output of the Greedy Algorithm



Output of the DSPP mechanism

- Greedy algorithm can cause massive reconfigurations
- Our DSPP algorithm is more adaptive

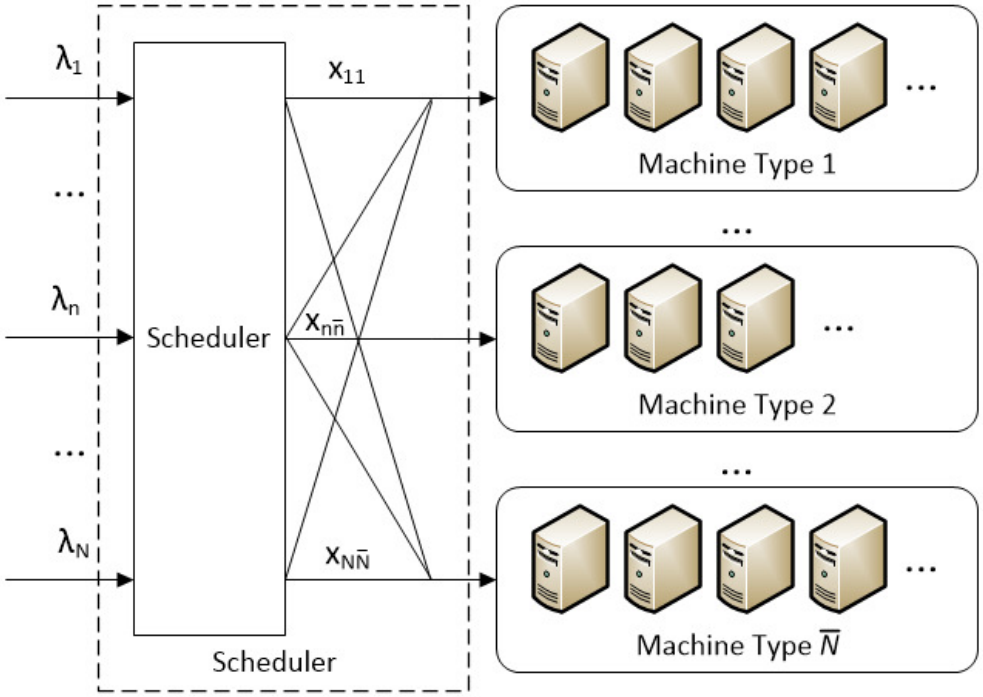
Outline

- Introduction
- Cloud Workload Management
- Research Contributions
  - Service Placement in Geo-Distributed Clouds
  - Heterogeneity-Aware Dynamic Capacity Provisioning
  - Fine-Grained Resource-Aware Scheduling for MapReduce
- Conclusion

Heterogeneity-Aware Capacity Provisioning

- Energy cost is an important concern in data centers
  - Accounts for 12% of operational cost [Gartner Report 2010]
  - Governments policies for building energy-efficient (i.e. “Green”) computing platform
- Dynamic Capacity Provisioning (DCP)
  - Minimize energy cost by turning off servers
  - An idle server consumes as much as 60% of its peak energy demand
- Limitations of existing work:
  - Lack of consideration to both machine and workload heterogeneity

Heterogeneity-Aware DCP



- Capture run-time workload composition
- Perform DCP at aggregate level to minimize impact on scheduling

Solution Approach

- Classify tasks based on their size and duration using *k-means* clustering algorithm
- Capture the run-time workload composition in terms of arrival rate for each task class
  - Predict the arrival rate of each type of tasks
- Define *container* as a logical allocation of resources to a task that belongs to a task class
- Use containers to reserve resources for each task class
  - Using task arrival rate to estimate the number of required containers of each type of task

Problem Formulation

$$\max_{\delta_t^m, \sigma_t^{mn}} \quad R_T = \sum_{t=1}^{T} U_t^{perf} - E_t - C_t^{sw}$$

- where

$$U_t^{perf} = \sum_{n \in N} f^n\Big(\sum_{m \in M} x_t^{mn}\Big) \qquad \text{(Performance objective)}$$

$$E_t = \sum_{m \in M} -p_t \left(z_t^m E^{idle,m} + \sum_{r \in R} \sum_{n \in N} \frac{\alpha^{mr} c^{nr}}{c^{mr}} \cdot x_t^{mn} \right) \qquad \text{(Energy cost)}$$

$$C_t^{sw} = \sum_{m \in M} q_m |\delta_t^m| \qquad \text{(Switching cost)}$$

- Subject to constraints

$$z_{t+1}^m = z_t^m + \delta_t^m \qquad \forall n \in N, m \in M, t \in \mathcal{T} \qquad \text{(Machine state constraint)}$$

$$x_{t+1}^{mn} = x_t^{mn} + \sigma_t^{mn} \qquad \forall n \in N, m \in M, t \in \mathcal{T} \qquad \text{(Workload state constraint)}$$

$$z_t^m \le N_t^m \qquad \forall m \in M, t \in \mathcal{T} \qquad \text{(Total capacity constraint)}$$

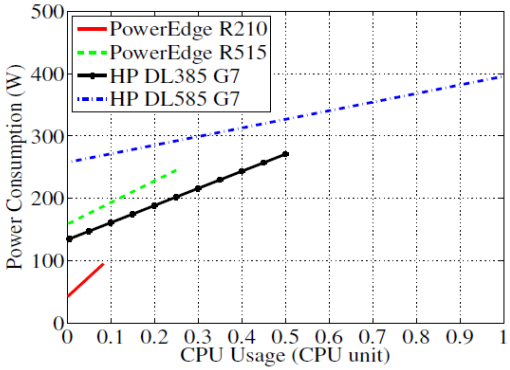
$$\sum_{n \in N} c_n^r x_t^{mn} \le z_t^m C^{mr} \qquad \forall m \in M, r \in R, t \in \mathcal{T} \qquad \text{(Capacity constraint)}$$

Solution Techniques

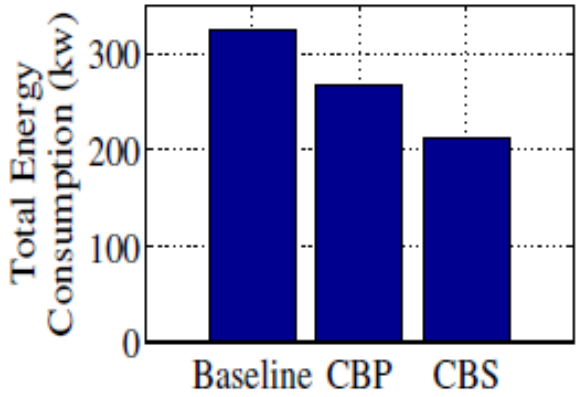
- Optimal Capacity Provisioning is ***NP-hard*** to solve

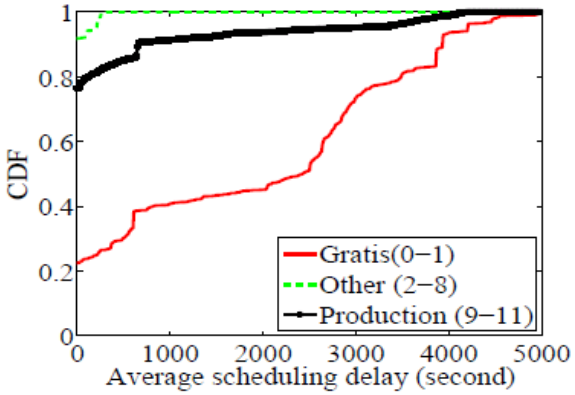
- We first solve the relaxation of the integer program, then we devise two approaches
  - Container-Based Scheduling (CBS)
    - Statically allocate containers in physical machines
    - At run-time, schedule tasks in containers
  - Container-Based Provisioning (CBP)
    - Use the estimated number of containers to provision machines
    - At run-time, schedule tasks using existing VM scheduling algorithms such as first-fit (FF)

Experiments

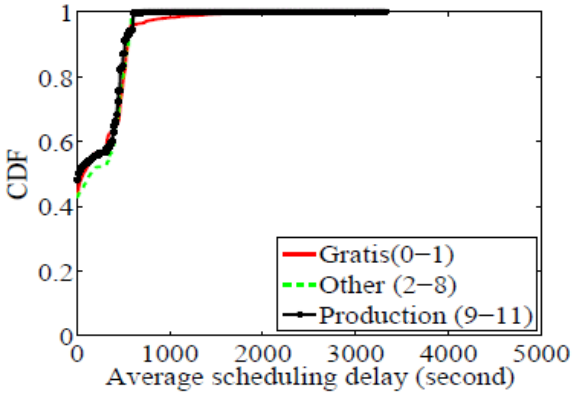


Machine Configurations

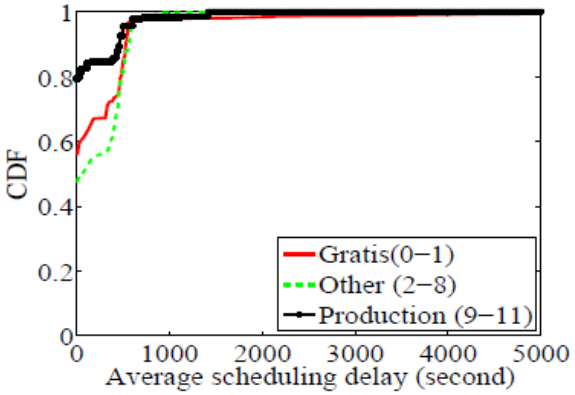




Baseline



CBP



CBS

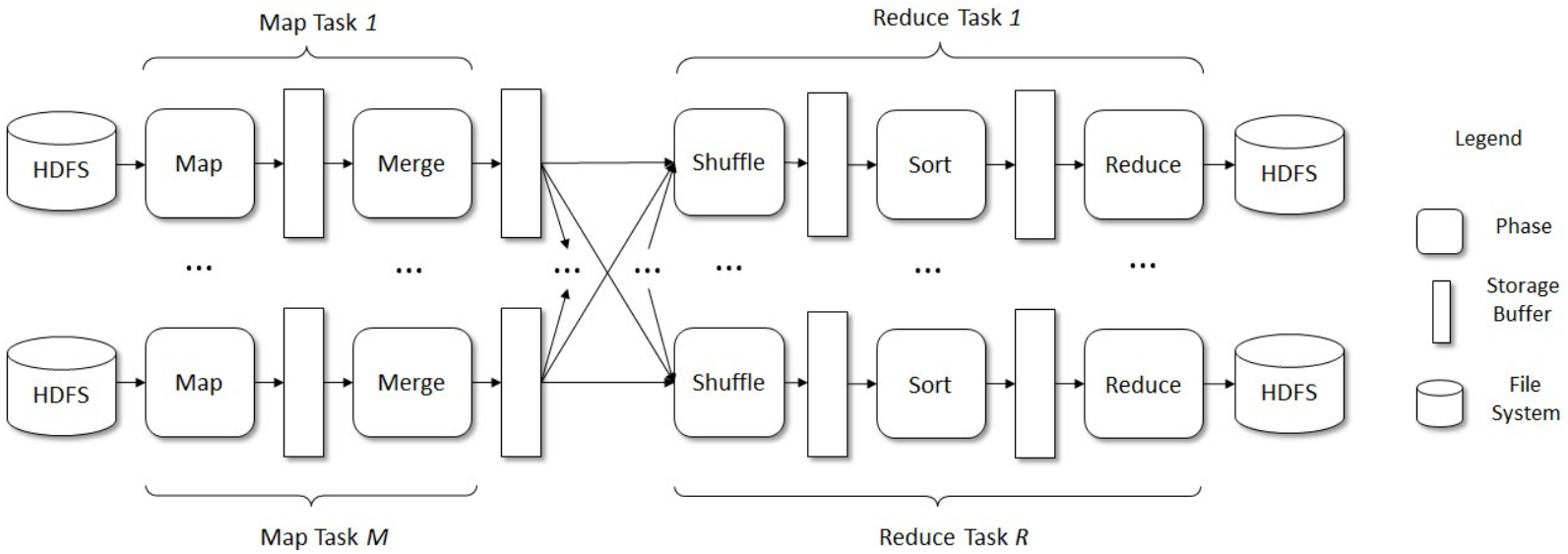
Outline

- Introduction
- Cloud Workload Management
- Research Contributions
  - Service Placement in Geo-Distributed Clouds
  - Heterogeneity-Aware Dynamic Capacity Provisioning
  - Fine-Grained Resource-Aware Scheduling for MapReduce
- Conclusion

Resource-Aware MapReduce Scheduling

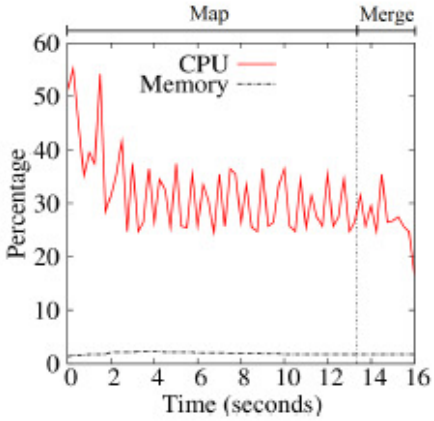
- MapReduce a popular framework for data intensive computations
  - Data sets are divided into blocks
  - Map tasks: processing individual blocks
  - Reduce tasks: aggregate Map outputs
- Resource-Aware scheduling is important
  - The original MapReduce adopts a slot-based allocation scheme
  - Hadoop v2 (a. k. a.) YARN is a resource-aware scheduler

Motivation

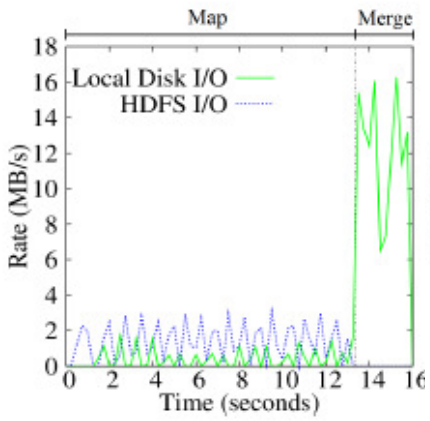


- Executing of a task can be divided into phases
- Phases have different resource characteristics
  - Shuffle is (network and disk I/O) intensive
  - Map and Reduce can be more CPU intensive

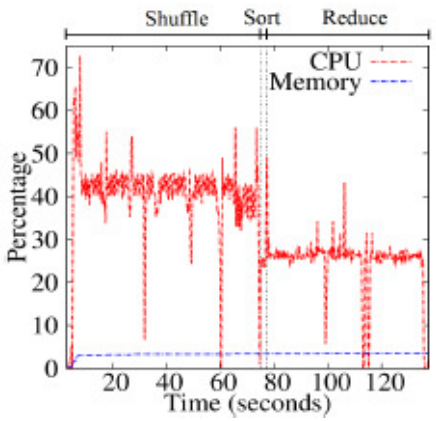
Motivation



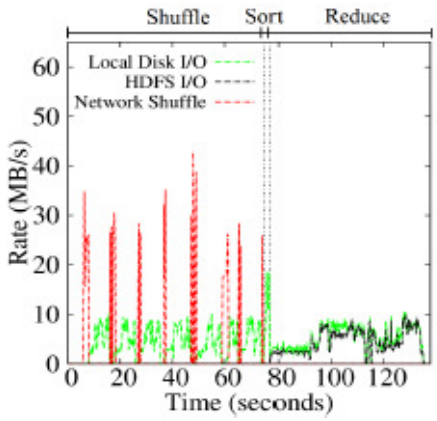
Map CPU/Mem Usage



Map I/O Usage



Reduce CPU/Mem Usage



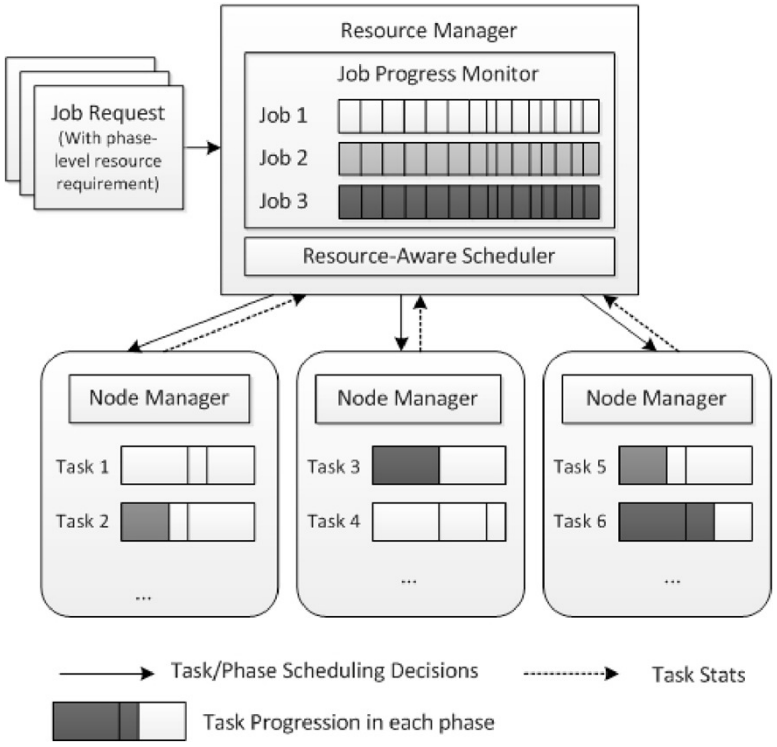
Reduce I/O Usage

Phase-Level Usage Characteristics of the InvertedIndex Job

- Observation: Phases have different Usage characteristics

PRISM

- PRISM is a fine-grained MapReduce Scheduler
  - Users can specify phase-level resource requirement as input
- At run-time, tasks request permissions to execute subsequent phases



Design Issues

- Design objectives
  - *Improving resource utilization:* Phase-level scheduling provides more "bin-packing" opportunities for improving utilization
  - *Avoiding resource contention:* Prioritize phase scheduling to make critical tasks run faster
- Design Considerations
  - *Fairness*: Every job should be getting sufficient resources over time to prevent starvation
  - *Performance:* Phases should not be delayed indefinitely to cause stragglers
    - Jobs with different deadlines should have different tolerance for delays

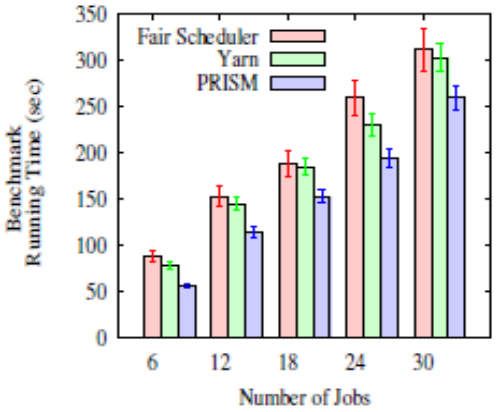
Scheduling Algorithm

- Greedy algorithm that schedule phases according to utilities
- The utility of scheduling a phase *i* on *n* is

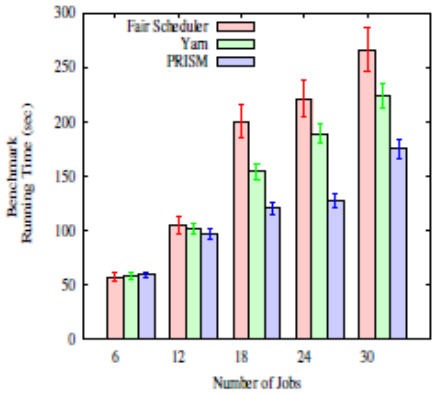
$$U(i,n) = U_{fairness}(i,n) + \alpha \cdot U_{perf}(i,n)$$

- where
  - $U_{fairness}(i,n)$ is the gain in fairness
  - $U_{perf}(i,n)$ denotes the gain in performance
    - If *i* is the starting phase of a task, $U_{perf}(i,n)$ denotes the gain in degree of parallelism
    - If *i* is the subsequent phase of a task, $U_{perf}(i,n)$ is an increasing function of the ratio between time paused and expected task completion deadline
- If additional resources are available, the idle resource is shared among tasks proportionally

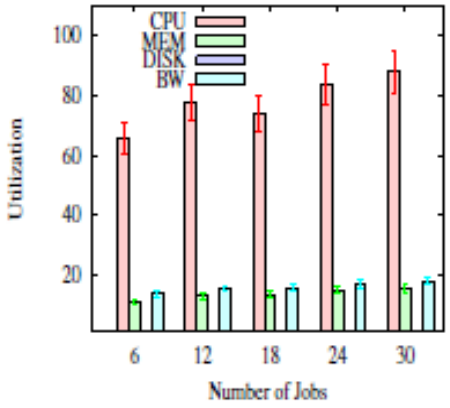
Experiments



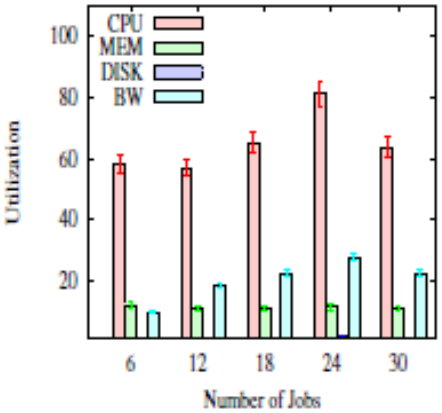
Job Running Time (PUMA)



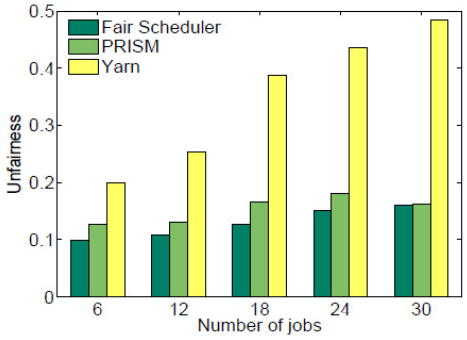
Job Running Time (Gridmix2)



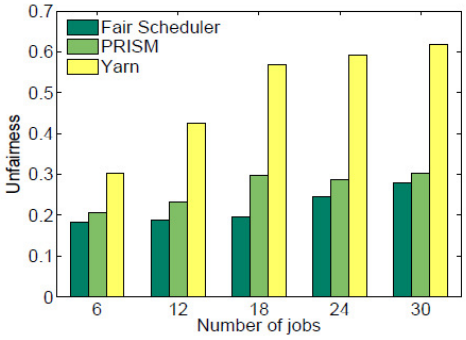
Utilization (Yarn)



Utilization (PRISM)



(a) Unfairness using PUMA



(b) Unfairness using Gridmix

Unfairness Result

Conclusion

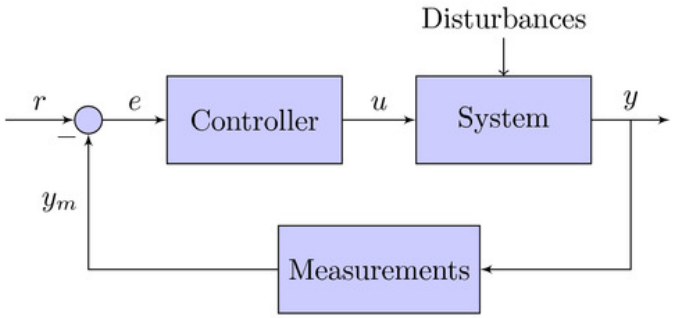
- Resource management is a major challenge of Cloud computing environment
- This thesis makes contribution on 3 specific challenges
  - Dynamic service placement in Geo-distributed Clouds
  - Heterogeneity-Aware Dynamic Capacity Provisioning in data centers
  - Fine-grained Resource-Aware MapReduce Scheduling

Thank you!

Service Placement Backup Slides

Why Model Predictive Control?

- Conventional feedback control (e.g. PID controller)
  - Do not handle ***general constraints*** (Only constraints on manipulated variables)
  - Unable to optimize the system if the overall system is non-square
  - Does not consider variable interactions in multi-variable (e.g. MIMO) systems
- MPC was introduced to address these limitations in the late 70's
  - Under mild conditions, MPC is guaranteed to achieve stability



A feedback control system

Service Reconfiguration Cost

- Reconfiguration cost
  - **Assumption**: Assuming VM images are already uploaded to the proper DCs
  - VM Start up time (time between launch and first successful ssh login) is dependent on data center, VM image size, operating system
  - Starting services may take longer

Table: VM startup Time (small instance: 1 core, 1.7GB memory)*

Cloud	OS	Average VM startup time
EC2	Linux	96.9 seconds
EC2	Windows	810.2 seconds
Azure	WebRole	374.8 seconds
Azure	WorkerRole	406.2 seconds
Azure	VMRole	356.6 seconds
Rackspace	Linux	44.2 seconds
Rackspace	Windows	429.2 seconds

- Shutting down VM may incur extra (bookkeeping) cost

* M. Mao and M. Humphrey, "A performance study on the VM Startup time in the Cloud" IEEE CLOUD, 2012

Service Reconfiguration Cost

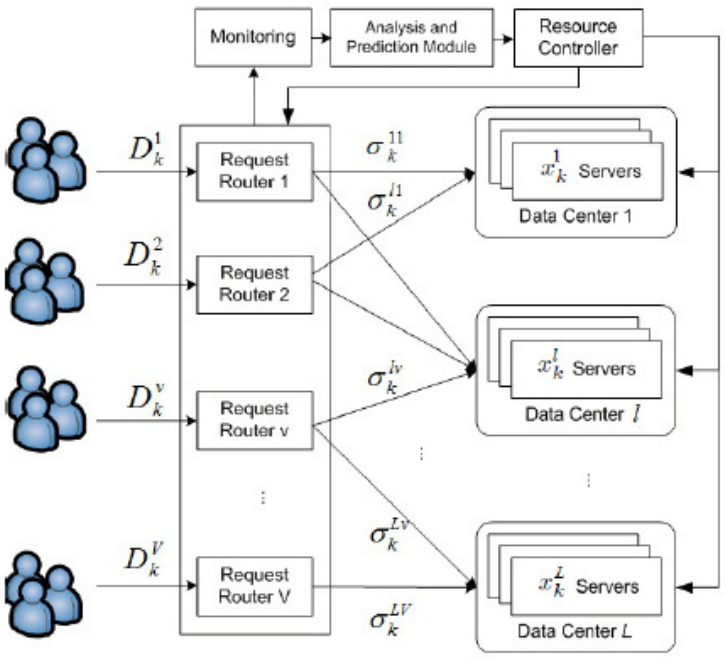
- Data storage model
  - Assume database servers are already running or the images are installed
    - Reconfiguration cost is mainly start-up time
  - Database servers may synchronize periodically
    - Future work

Related Work

- Replica placement
  - Most of the work studies on static problem where demand is not changing
  - Heuristic algorithm (e.g. local search) for dynamic cases
- Service placement
  - Energy and carbon footprint-aware placement (e.g. FORTE[1], Rao[2] and Liu[3])
- **To be best of our knowledge, reconfiguration cost was not studied before our work**

[1]X. Gao, A. Curtis, B. Wong, and S. Keshav. "It's not easy being green." ACM SIGCOMM, 2012
[2]L. Rao, X. Liu, and W. Liu. "Minimizing electricity cost: Optimization of distributed internet data centers in a multi-electricity-market environment", IEEE INFOCOM, 2010
[3]Z. Liu, M. Lin, A. Wierman, S. Low, and L. Andrew. Greening geographical load balancing. ACM SIGMETRICS, 2011

Service Placement Model (Single Tier, Single Provider)



Service Placement Model

Minimize

$$J := \sum_{k=0}^{K} H_k + G_k + P_k + C_k$$

Where

$$H_k = \sum_{l \in L} \sum_{v \in V} x_k^{lv} p_k^l \quad \text{(Resource cost)}$$

$$G_k = \sum_{l \in L} c_{on}^l (\sum_{v \in V} u_k^l)^+ - c_{off}^l (\sum_{v \in V} u_k^l)^- \quad \text{(Reconfiguration Cost)}$$

$$P_k = \sum_{v \in V} h(\sum_{l \in L} \frac{x_k^{lv}}{a^{lv}} - D_k^v) \quad \text{(Performance cost)}$$

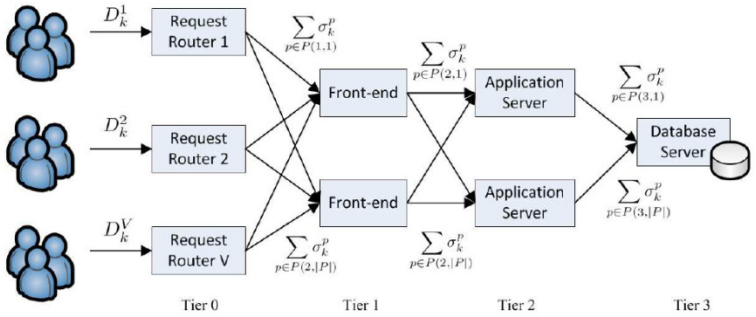
$$C_k = \sum_{l \in L} \sum_{r \in R} \pi^r (s^r \sum_{v \in V} x^{lv} - C^r)^+ \quad \text{(Capacity cost)}$$

Subject to

$$x_{k+1}^{lv} = x_k^{lv} + u_k^{lv}, \quad \forall l \in L, v \in V, 0 \le k \le K \quad \text{(State equation)}$$

$$x_k^{lv} \ge 0 \qquad \forall l \in L, v \in V, 0 \le k \le K$$

Service Placement Model (N-Tier service, Single Provider)



Minimize $J := \sum_{k=0}^{K} H_k + G_k + P_k + C_k$

Where

$$H_k = \sum_{n=1}^{N} \sum_{l \in L} \sum_{p \in P} x_k^{pnl} p_k^{nl}$$ (Resource cost)

$$G_k = \sum_{n=1}^{N} \sum_{l \in L_n} c_{on}^{nl} \Big(\sum_{p \in P(n,l)} u_k^{pnl} \Big)^+ - c_{off}^{nl} \Big(\sum_{p \in P(n,l)} u_k^{pnl} \Big)^-$$ (Reconfiguration Cost)

$$P_k = \sum_{v \in V} \sum_{n \in N} h\Big(\sum_{l \in L} \sum_{p \in P(0,v) \cup P(n,l)} \frac{x_k^{pnl}}{a_k^{pnl}} - D_k^v \Big)$$ (Performance cost)

$$C_k = \sum_{r \in R} \pi^r \Big(s^r \sum_{n=1} \sum_{l \in L} \sum_{p \in P(n,l)} x_k^{pnl} - C^r \Big)^+$$ (Capacity cost)

Subject to

$$x_{k+1}^{pnl} = x_k^{pnl} + u_k^{pnl} \qquad \forall l \in L, v \in V, 0 \le k \le K+1$$ (State equation)

$$x_k^{pnl} \ge 0 \qquad \forall l \in L, v \in V, 0 \le k \le K$$

Handling Unexpected Spikes

- Underestimation of future demand (e.g., unexpected demand spikes) can lead to under-provisioning of server resources.
- There are several possible ways to deal with this limitation.
  - Using an overprovisioning factor
    - Padding to handle risks
  - Faster reconfiguration rate
    - Reasonable since reconfiguration cost is considered

Analysis

- Price of Stability (PoS) = Best outcome / optimal outcome
- Price of Anarchy (PoA) =Worst outcome / optimal outcome
- Theorem 1: Assume that the prediction horizon of each SP *i* is the same, then the price of stability (PoS) is 1
  - Proof idea: Optimal outcome with a given prediction horizon is a stable outcome for all SPs
- Theorem 2: Assume that the prediction horizon of each SP *i* is the same, then the price of anarchy (PoA) is infinity
  - Proof idea: Hard capacity constraint and high performance penalty can make the solution arbitrarily worse

Mechanism

- Using a price-driven mechanism using dual-decomposition
  - Each SP uses the current price to control service using MPC:
  - Based on resource demand, Cloud provider computes a congestion price

$$\min_{\mathbf{u}^i_{k+t|k}} \sum_{t=0}^{W} \sum_{v \in V} (\mathbf{e}^{\top}_{k+t|k} \mathbf{s}^i + \lambda_{k+t|k} \mathbf{s}^i) \mathbf{x}^{iv}_{k+t|k} + \mathbf{R}^{i\top} g^i_k(\mathbf{u}^i_{k+t|k}) + P^i_k(\mathbf{a}^i_{k+t} \mathbf{x}^i_{k+t|k} - \mathbf{D}^i_{k+t|k})$$

  - And update price for future periods incrementally

$$\min_{\mathbf{v}_{k+t|k} \in \mathbb{R}^V} \quad \sum_{t=0}^{W} \pi(\mathbf{v}_{k+t|k}) - \lambda_{k+t|k}(\mathbf{v}_{k+t|k})$$

$$\lambda_{k+t|k} := (\lambda_{k+t|k} + \alpha(\sum_{i \in \mathcal{N}} \sum_{v \in V} \mathbf{s}^i \mathbf{x}^i_{k+t|k} - \mathbf{v}_{k+t|k}))_+$$

Experiments

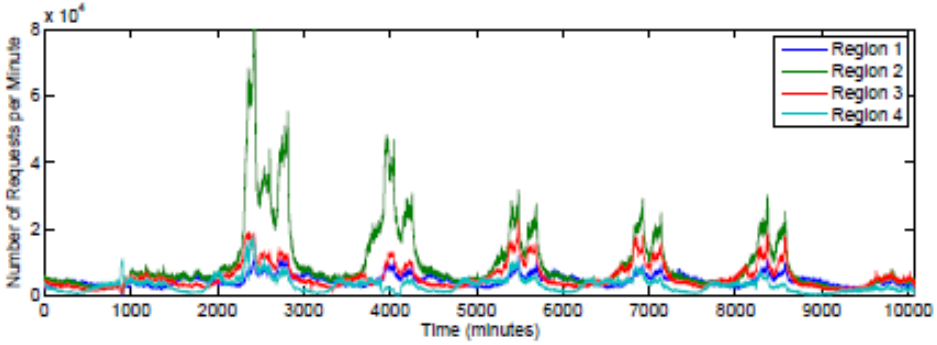


Figure 3.5: HTTP requests in the Worldcup 98 dataset

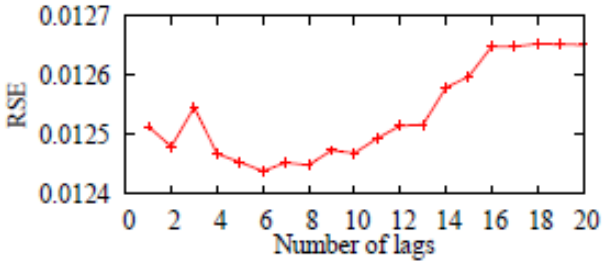


Figure 3.6: Prediction Accuracy vs. number of lags used

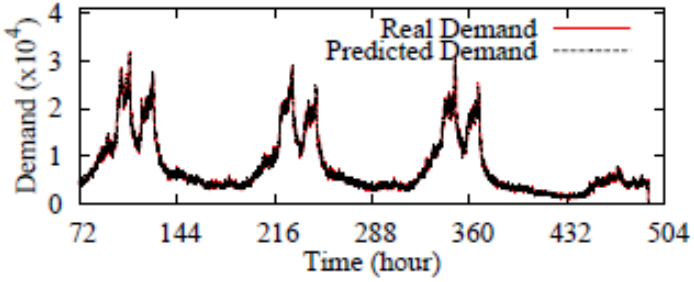


Figure 3.7: Actual vs. Predicted demand for region 2

Experiments

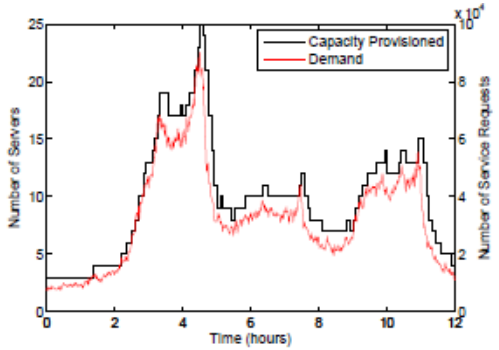


Figure 3.8: Response to Demand Fluctuation

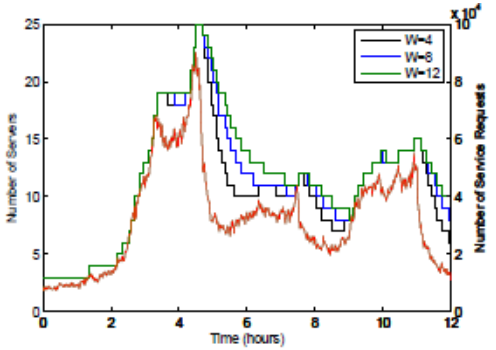


Figure 3.9: Effect of prediction window size on the number of servers

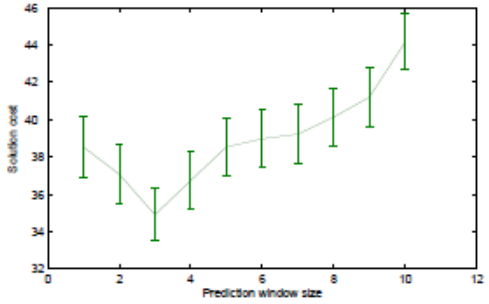


Figure 3.10: Effect of prediction horizon on the solution cost

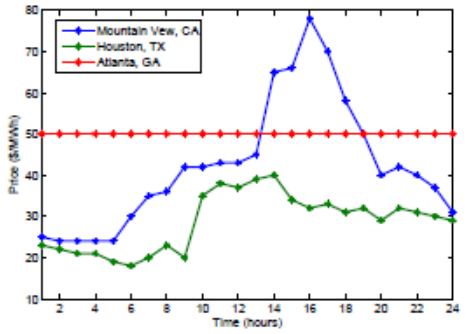


Figure 3.11: Prices of electricity used in experiments

Experiments

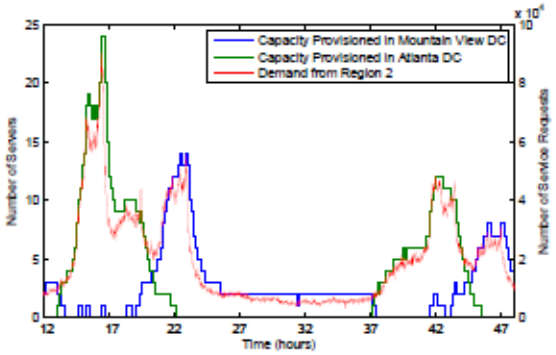


Figure 3.12: Impact of Price on Solution quality

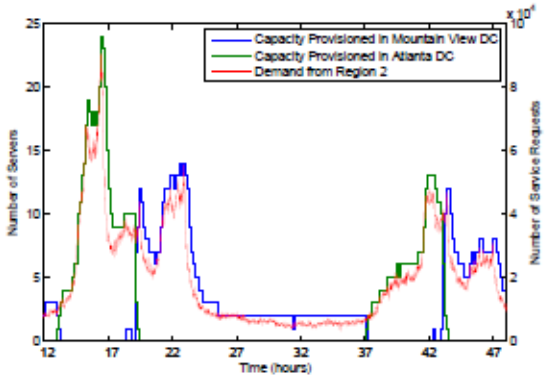


Figure 3.13: Output of the greedy algorithm

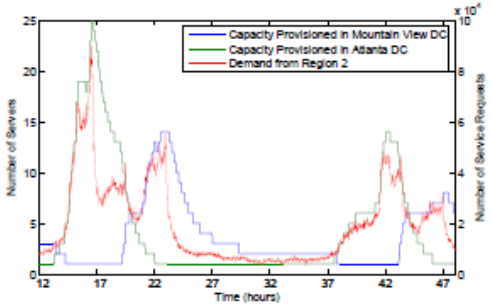


Figure 3.14: Output with no reconfiguration cost and long window size

Multi-Player Results

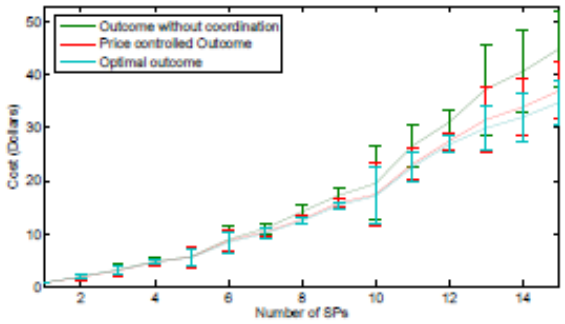


Figure 3.15: Comparing the cost of NEs versus the number of players

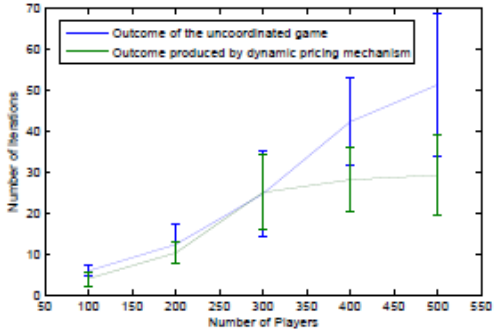


Figure 3.16: Number of players vs. convergence rate

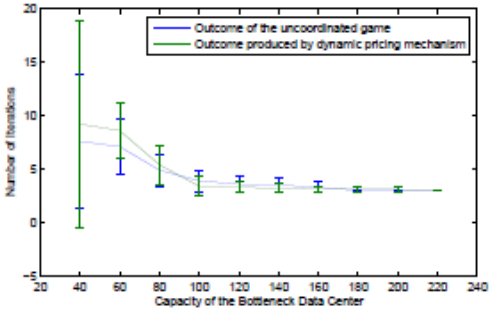


Figure 3.17: Capacity of the Bottleneck DC vs. convergence rate

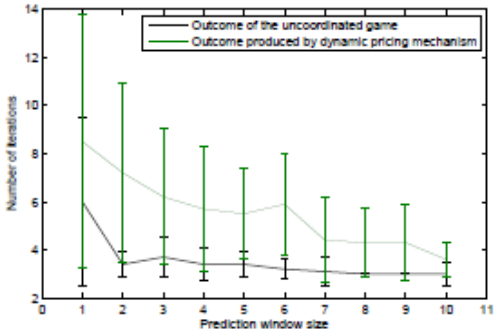


Figure 3.18: Prediction horizon length vs. convergence rate

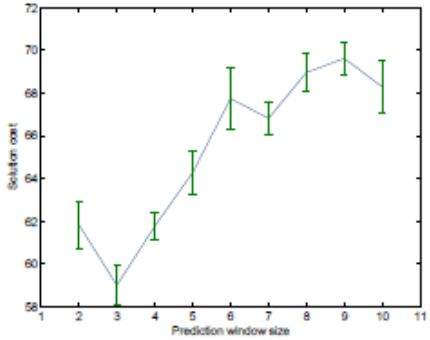


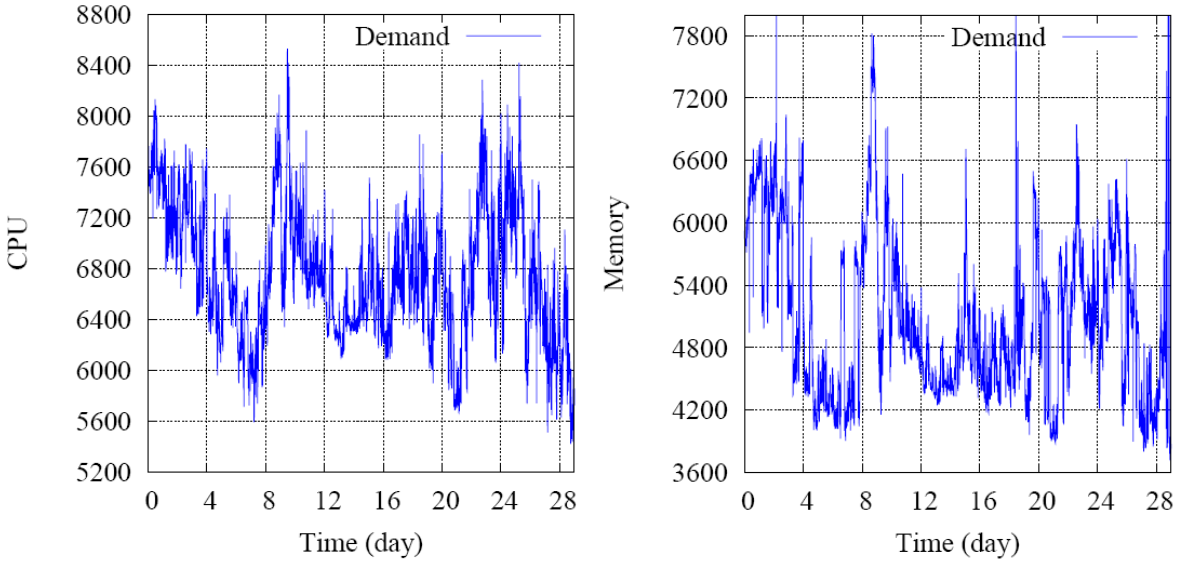
Figure 3.19: Impact of prediction horizon length on the cost

Harmony

Trace Analysis

- Workload traces collected from a production compute cluster in Google over 29 days
  - ~ 12,000 machines
  - ~2,012,242 jobs
  - 25,462,157 tasks
- Applications are represented by **jobs**
  - **User-facing jobs**: e.g., 3-tier web applications
  - **Batch jobs:** e.g., MapReduce jobs
- Each job consists of one or more **tasks**
- There are 12 priorities that are divided into three priority groups: gratis(0-1), other(2-8), production(9-11)

Trace Analysis: Total Resource

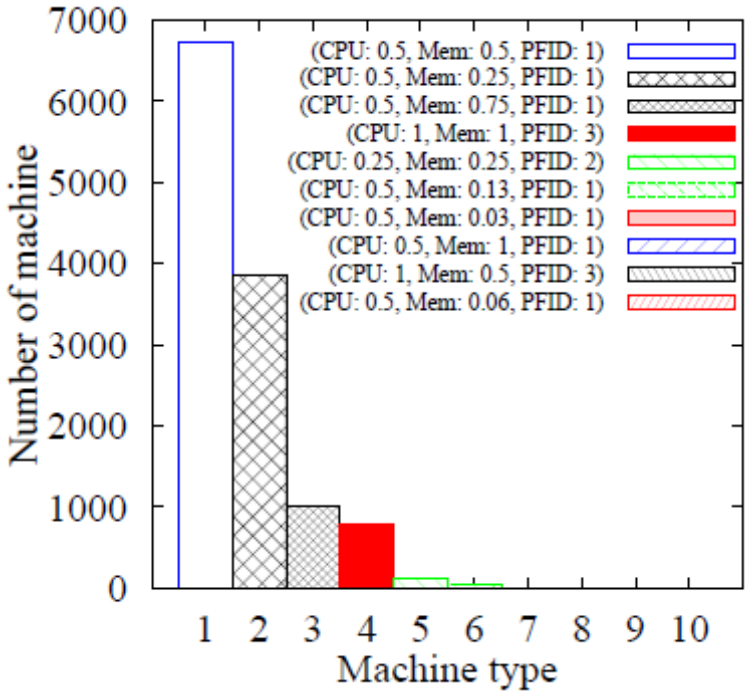


CPU Demand over 30 days

Memory Demand over 30 days

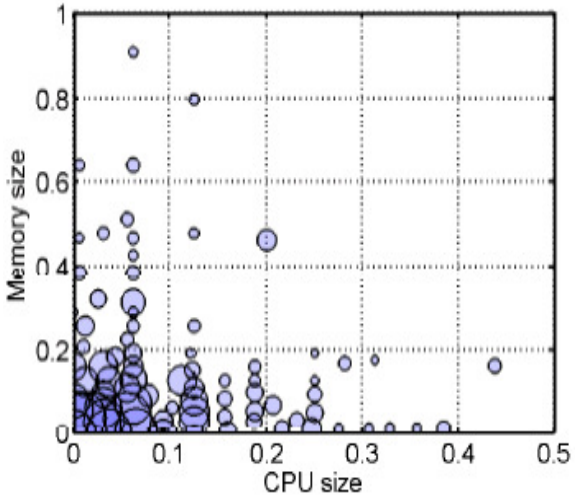
Figure: Total resource demand in Google's Cluster Data Set

Machine Heterogeneity

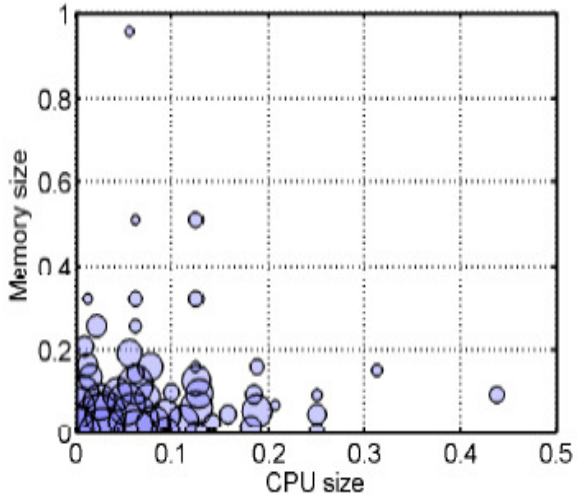


- 10 types of machines, some (e.g type 2 and 4) have high CPU capacity, others have high memory (e.g type 3 and 8) capacity

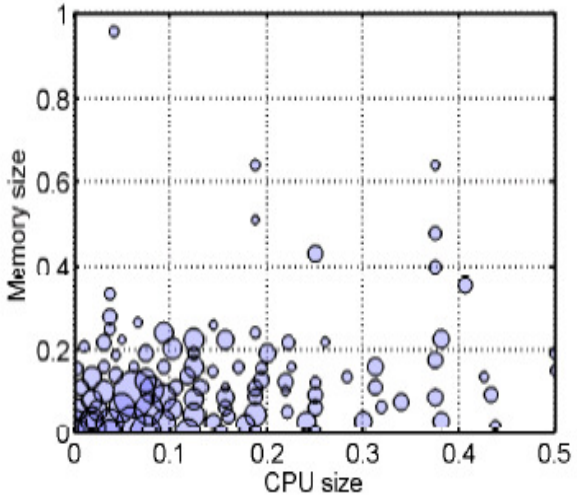
Trace Analysis: Task Size



(a) Gratis (0-1)



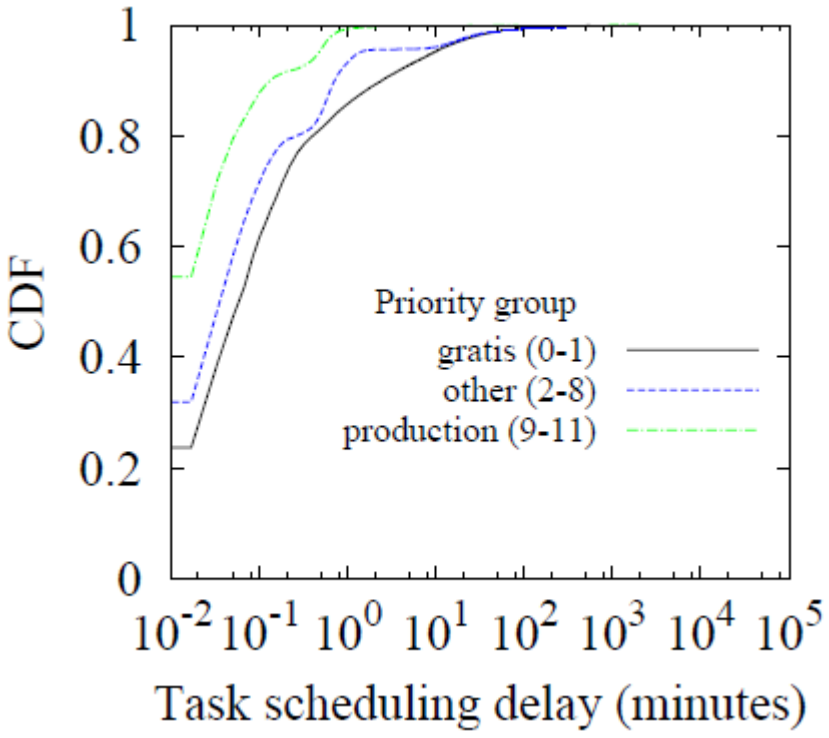
(b) Other

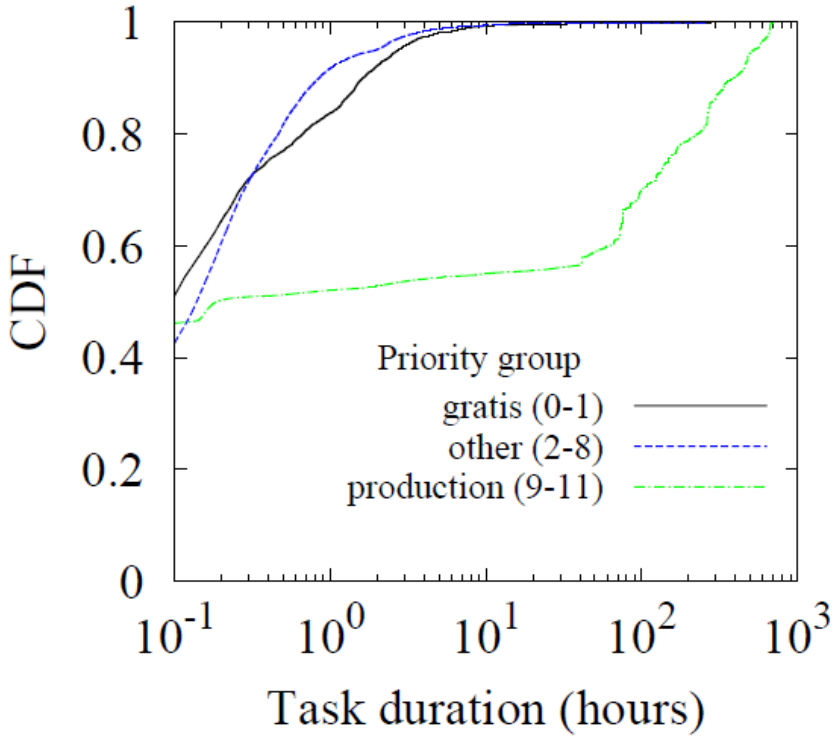


(c) Production

- Tasks are either CPU intensive or Memory intensive
- Little correlation between CPU size and Memory size

Task Priority and Running Time





Summary

- Machines have different resource capacities
  - Some have more CPU capacities, while others have more memory capacities
- Tasks belong to different jobs have different resource requirements, running time and priorities
- Heterogeneity-awareness is important
  - Different machines are likely to have different energy characteristics
  - Not every task can be scheduled on every machine

Handling Unexpected Spikes

- Underestimation of future demand (e.g., unexpected demand spikes) can lead to under-provisioning of server resources.
- There are several possible ways to deal with this limitation.
  - Using an overprovisioning factor
    - Padding to handle risks
  - Faster reconfiguration rate
    - Reasonable since reconfiguration cost is considered

Reconfiguration Costs

- Wear-and-Tear cost is 0.5 cents per power-cycle
- Reason
  - wear-and-tear effect mainly affects disk reliability
  - The typical MTBF for disk is around 40000 on/off cycles.
  - Therefore, assuming a disk failure costs around $200 (including labor fee), then the cost for an on-off cycle is roughly $200/40000 = 0.5 cents, as suggested by [1][2]

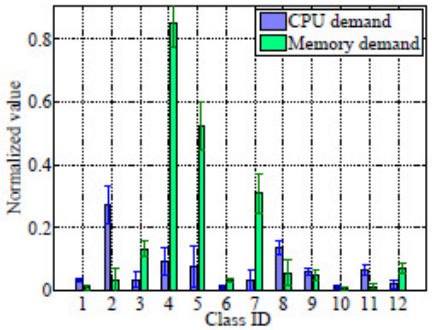
[1] Managing Server Energy and Operational Costs in Hosting Centers, SIGMETRICS 2005
[2] A Case For Adaptive Datacenters To Conserve Energy and Improve Reliability, UC Berkeley Technical Report, 2008

CBS Scheduling Details

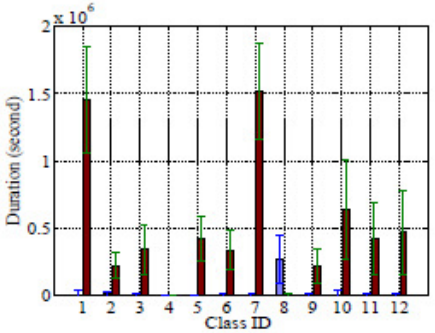
- The scheduling algorithm essentially schedules each task in the first available slot that has sufficient capacity to run it, other $\prod_{m \in M^k} \Pr(\exists r : \sum_{j \in N} s^{jr} + s^{ir} > C^{mr} | \sum_{j \in N} c^{jr} + c^{ir} \leq C^{mr}) \leq \epsilon$ e scheduling queue (What discipline is it?)
- **Theorem 4.** *Assume each task s^{kr} in each class k is independently and identically distributed with mean μ^{kr} and standard deviation σ^{kr} for each resource type r. Also, let M^k denote the minimum number of machines on which a type k task can be scheduled. We can set container size of task type k to* at

$$c^{kr} = \mu^{kr} + \sqrt{\frac{|R| - \epsilon^{\frac{1}{M^k}}}{\epsilon^{\frac{1}{M^k}}}} \sigma^{kr}$$

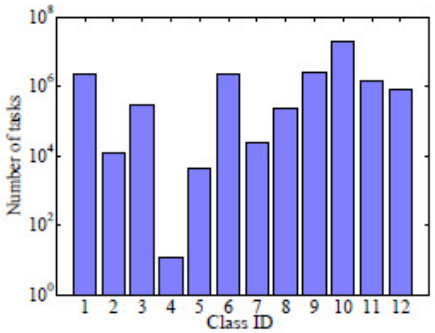
Experiments (Clustering)



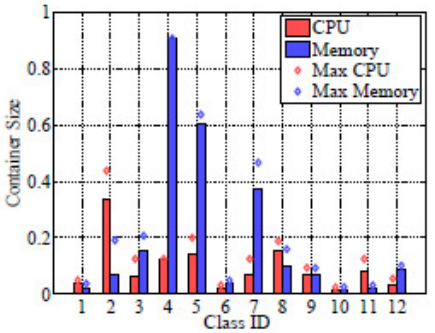
Class size (gratis)



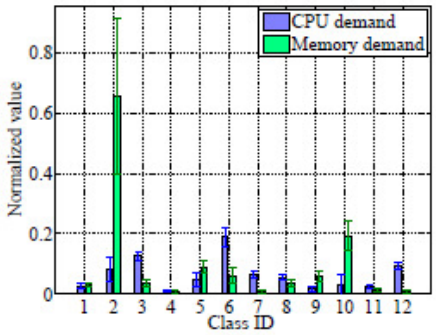
Duration (gratis)



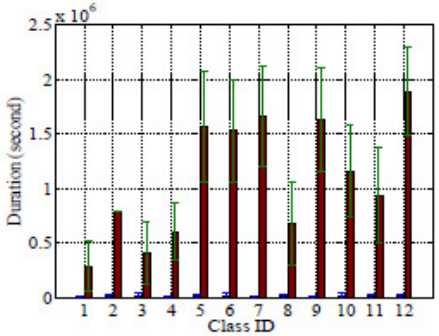
Number of tasks (gratis)



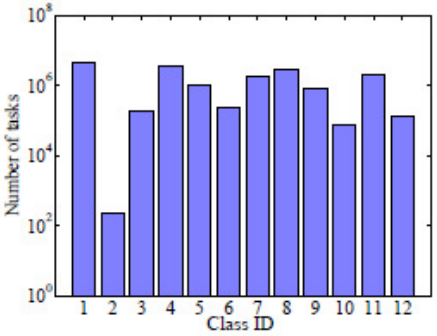
Container size (gratis)



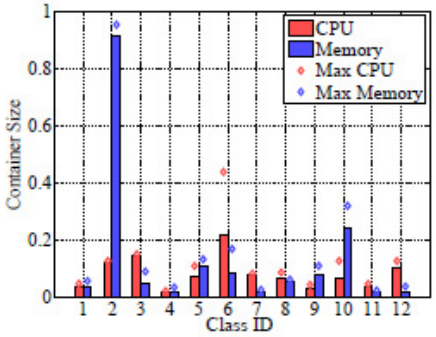
Class size (Other)



Duration (Other)

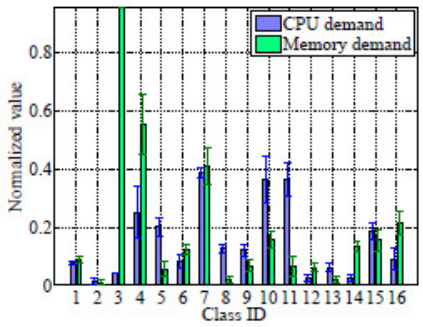


Number of tasks (Other)

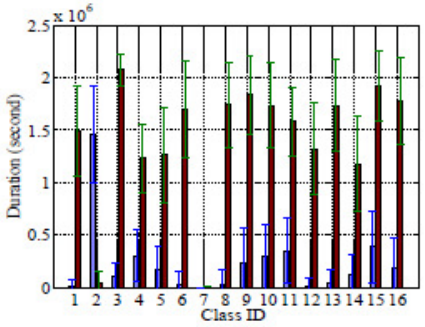


Container size (Other)

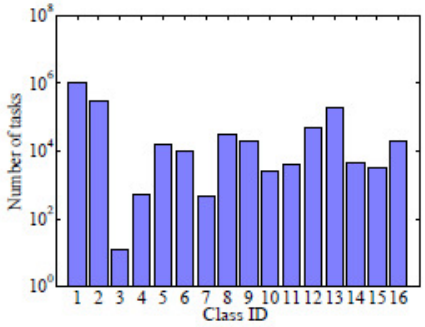
Experiments (Clustering)



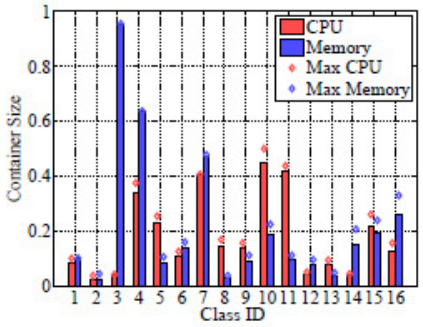
Class size (Production)



Duration (Production)

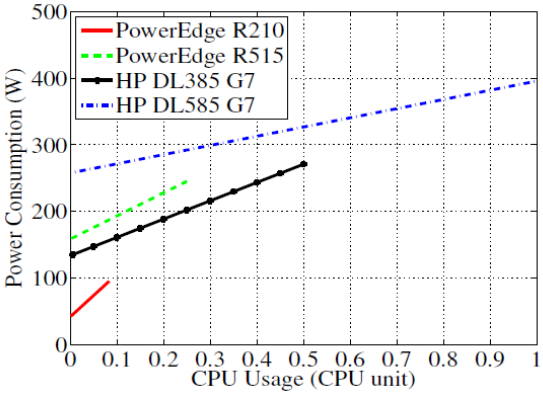


Number of tasks (Production)

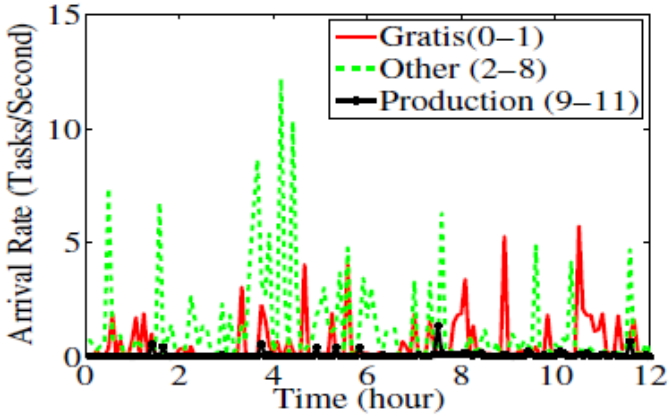


Container size (Production)

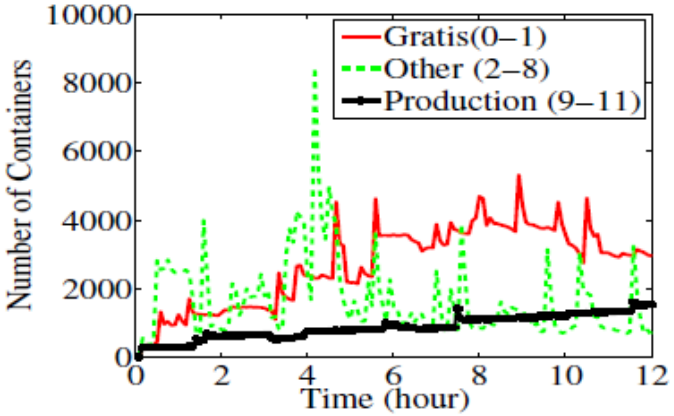
Experiments



Machine configurations

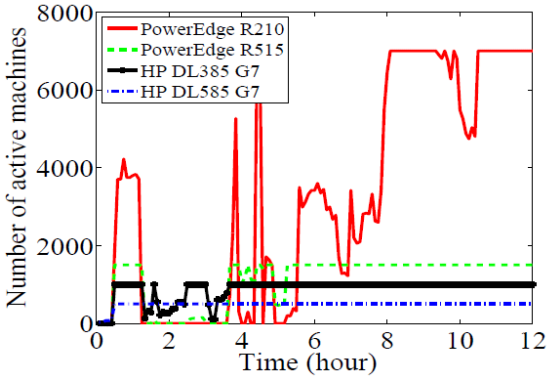


Aggregated task arrival rates

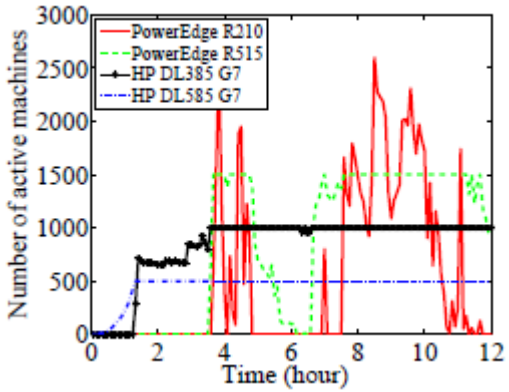


Number of required containers

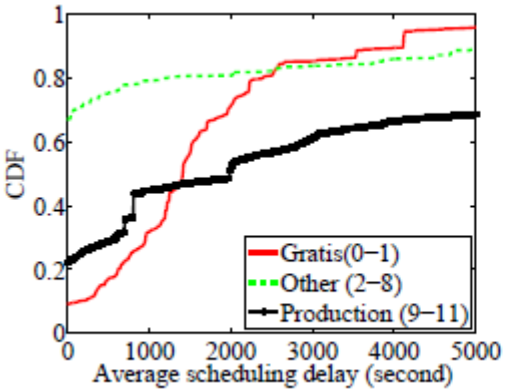
Experiments



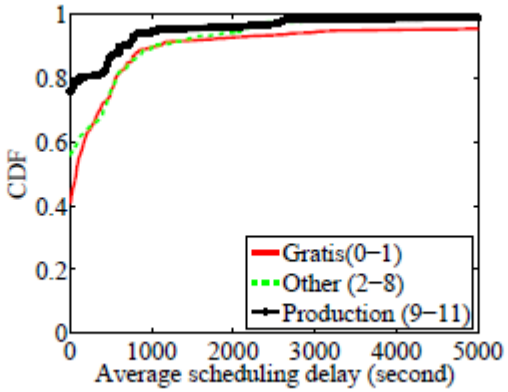
Baseline



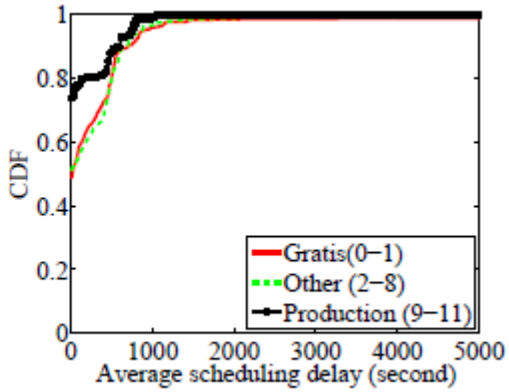
CBP/CBS



Baseline

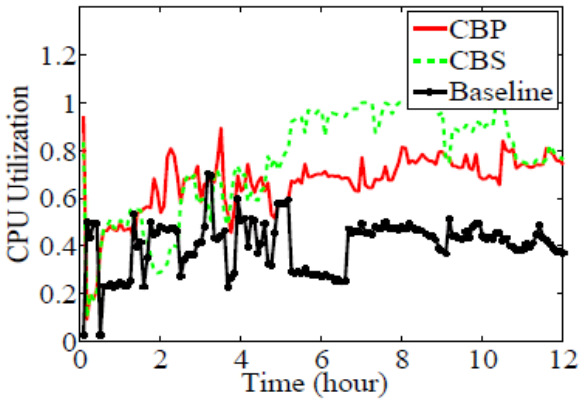


CBS

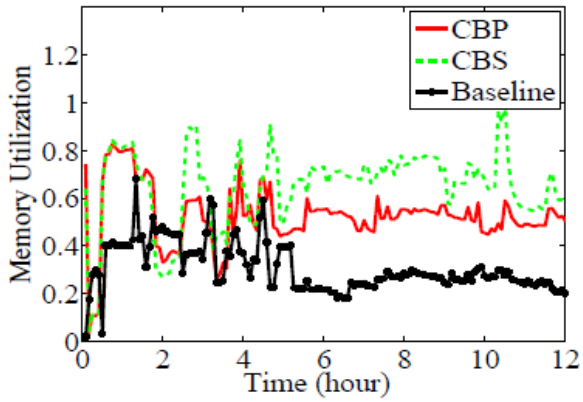


CBP

Experiments

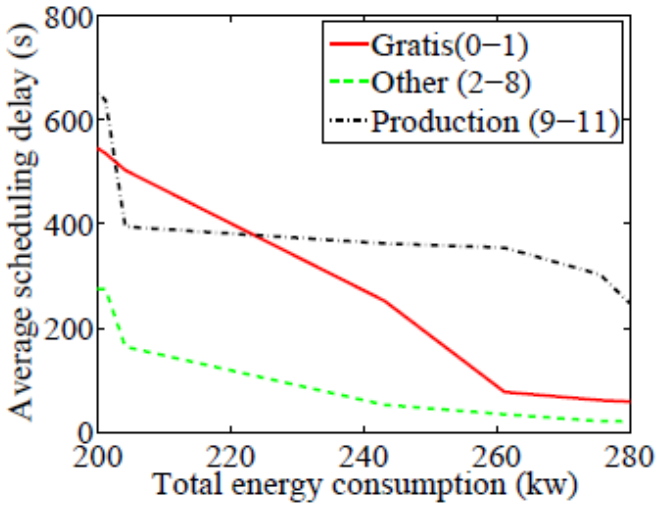


CPU Utilization

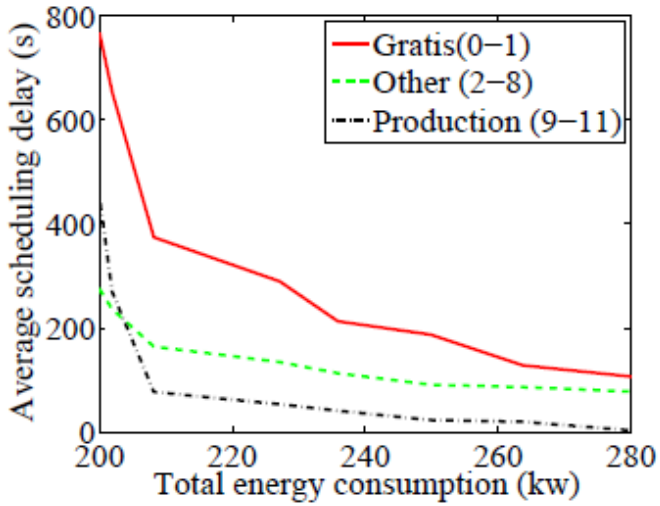


Memory Utilization

Comparing CBS and CBP



CBP



CBS

Trade-off curve obtained by varying the over provisioning factor (1-2)

- CBS generally performs better than CBP especially for production priority, due to guaranteed slots
- However, when underestimation occurs (very low overprovisioning factor), CBS performs worse.

PRISM

Theoretical Background

- Scheduling is studied under the field of ***scheduling theory***
  - MapReduce scheduling is similar to ***Job-shop scheduling***
    - Tasks may or may not have sequential dependencies
    - Online version with multiple machines is NP-hard
      - List scheduling is a well known competitive scheduling algorithm for job-shop scheduling
  - Differences between MapReduce scheduling and Job-shop scheduling
    - Multiple types of resources with heterogeneous resource capacities
      - Bin-packing algorithms are more appropriate

Related Work

- Hadoop v2 (a.k.a. Yarn)
  - Specifying resource requirement at task-level
- Resource Aware Scheduler (RAS)
  - Profile driven, but not phase-level
- MROrchstrator
  - Reconcile run-time resource contention
- Overlapping shuffle and reduce phases
  - considered the different resource consumption characteristics of each phase at run-time are not considered

Fairness metrics

- Running-time fairness (Fair scheduler and Quincy)
  - Equalize the speed up / slow down of each job

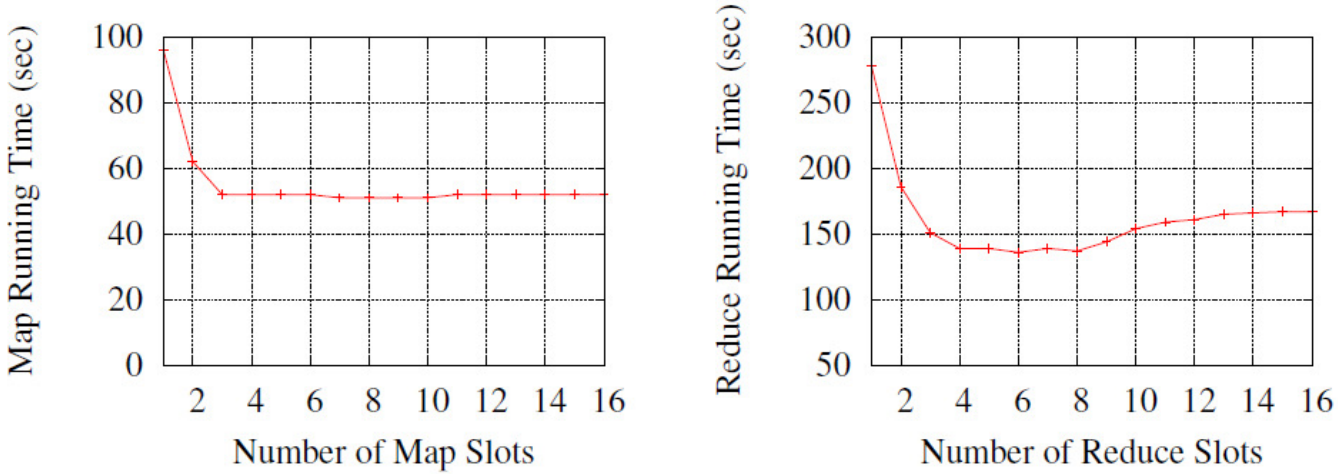
$$\frac{n_1}{N_1} = \frac{n_2}{N_2} = \ldots = \frac{n_J}{N_J}$$

- Dominant Resource Fairness (Supported in Yarn)
  - Equalize the share of each job/user's most demanded resources

$$\max_{r \in R}\left\{\frac{c_{1r}}{C_r}\right\} = \max_{r \in R}\left\{\frac{c_{2r}}{C_r}\right\} = \ldots = \max_{r \in R}\left\{\frac{c_{Jr}}{C_r}\right\}$$

Generating Profiles

- We vary the number of slots in the fair scheduler
  - First vary the number of Map slots to find optimal map stage running time
  - Once num. of map slots are fixed, vary the number of reduce slots find optimal job running time



Profiling for the Sort job

Other Issues

- PRISM is implemented based Hadoop Fair scheduler, thus it inherits many properties from Fair Scheduler
- Scalability:
  - The execution of the algorithm takes <50ms
- Sensitivity:
  - We found the profiles to be stable, as confirmed in the previous work (ARIA)*
- Failures:
  - Greedy algorithm naturally supports fault tolerance
- Speculative re-execution
  - Supports speculative re-execution just like fair scheduler

*ARIA: Automatic Resource Inference and Allocation for MapReduce Environments, ICAC 2011

Other Issues

- Tuning the various parameters for the utility function and make trade-offs between fairness/stragglers and concurrency.
  - Based on experience (future work)
- What about general purpose workloads, data skew, etc?
  - Data locality can be considered by profiling local and non-local tasks
- Missing details/ unstated assumptions for job profiling, would be good to add to the technical content.
  - Can use general profilers that profile at phase-level. Designing of the profiler is beyond the scope of this work

Other Issues

- phase-level scheduling for fault tolerance or speculative execution.
  - Can support speculative re-execution (to be added)
- would have appreciated a discussion of how this additional information could be easily estimated and provided to PRISM.
- Benchmarks of small job performance and scalability of the JT would have been useful.
  - Indeed, we are looking for hierarchical scheduling algorithms (future work)

Other Issues

- The experimental results appear to use "average job completion time" as the main metric. I find this unsatisfactory as a metric. The CDF is really important. Is there starvation? Do the proposed strategies really improve some jobs (e.g., small ones) at the cost of few others (e.g., large jobs)?
  - Variation of job running time is reported using unfairness metric
- Did the experiments turn speculative execution on?
  - Yes, it is turned on.