Resource Allocation Using Virtual Machines

November 6, 2013

Clusters

- Traditional HPC:
  - Supercomputing: 81% of Top500 list
  - Many “small” clusters everywhere
- Data Processing:
  - Google: 500,000 clustered servers, Map/Reduce operations
  - Hadoop, Dryad, etc.
- Service Hosting:
  - IBM’s Blue Cloud
  - Amazon Elastic Cloud (EC2)

Cluster Costs

- Startup Costs:
  - At least N times a single computer
  - Large clusters require a server room
  - “Enterprise” hardware
  - Dedicated cooling systems
  - Professional installation
- Ongoing Costs:
  - Electricity (extra for cooling)
  - Repairs
  - Dedicated staff

Cluster Sharing

- So, clusters are shared.
  - To spread costs
  - To keep utilization levels high
- Current scheduling approaches for HPC clusters
  - Gang scheduling
  - Batch scheduling

Gang scheduling: which no one uses

- Globally coordinated time sharing
    - Related threads or processes scheduled to run simultaneously on different processors
    - Coordinated context switching is performed across all nodes to switch from the processes in one time-slice to those in the next time-slice
- Complicated and slow
- Memory pressure a concern

Batch scheduling: which no one likes

- Usually FCFS with backfilling
- Backfilling needs (unreliable) compute time estimates
  - User predictions are inaccurate even in the presence of strong incentives for accuracy
  - Automatically predicted execution times may be more accurate than user provided ones
  - Using backfilling with estimates equal to twice the actual execution times improve performance
- No particular objective
- User dissatisfaction
- Inefficient use of nodes/resources

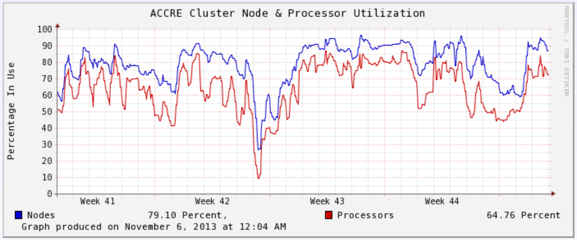
User Dissatisfaction

- Known disconnect between user satisfaction and what job schedulers do [Lee and Snavely, 2007]
- What's needed: a sound objective metric
- Difficult to optimize
- Easier with preemption and migration

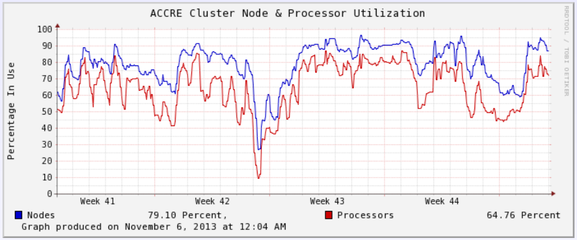
First Example of Cluster Usage

Advanced Computing Center for Research & Education
Vanderbilt University
30 days usage up to November 5, 2013
`http://www.accre.vanderbilt.edu/`

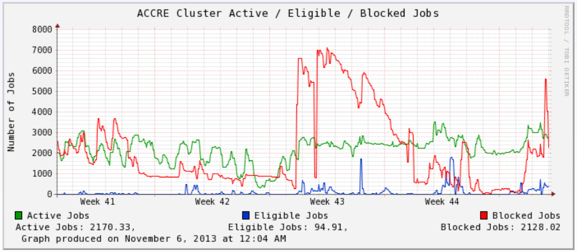
First Example of Cluster Usage



First Example of Cluster Usage

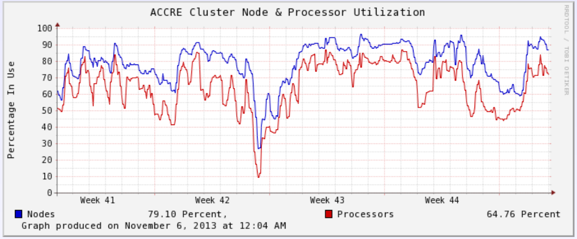


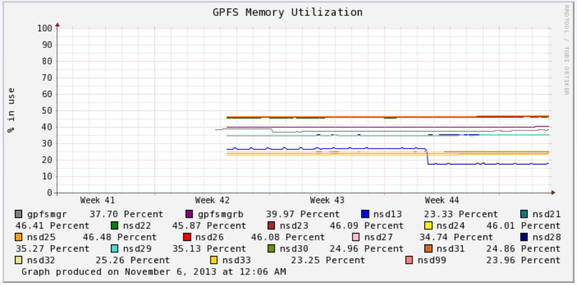
🟦 Nodes 79.10 Percent, 🟥 Processors 64.76 Percent
Graph produced on November 6, 2013 at 12:04 AM



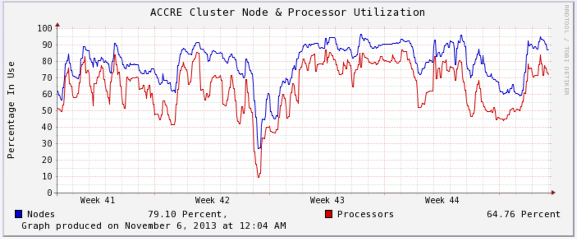
🟩 Active Jobs 🟦 Eligible Jobs 🟥 Blocked Jobs
Active Jobs: 2170.33, Eligible Jobs: 94.91, Blocked Jobs: 2128.02
Graph produced on November 6, 2013 at 12:04 AM

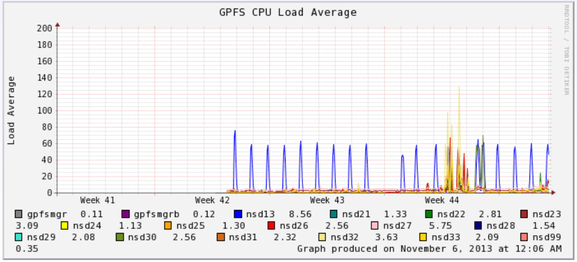
First Example of Cluster Usage



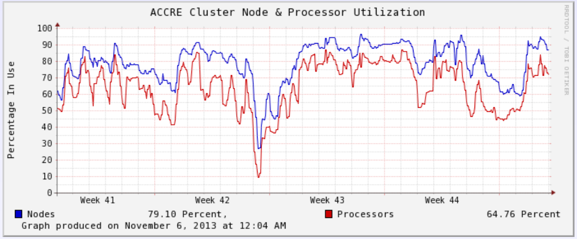


First Example of Cluster Usage



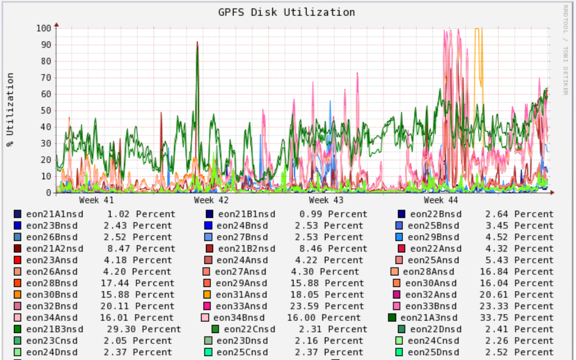


First Example of Cluster Usage



🟦 Nodes	79.10 Percent,	🟥 Processors	64.76 Percent

Graph produced on November 6, 2013 at 12:04 AM



eon21A1nsd	1.02 Percent	eon21B1nsd	0.99 Percent	eon22Bnsd	2.64 Percent
eon23Bnsd	2.43 Percent	eon24Bnsd	2.53 Percent	eon25Bnsd	3.45 Percent
eon26Bnsd	2.52 Percent	eon27Bnsd	2.53 Percent	eon29Bnsd	4.52 Percent
eon21A2nsd	8.47 Percent	eon21B2nsd	8.46 Percent	eon22Ansd	4.32 Percent
eon23Ansd	4.18 Percent	eon24Ansd	4.22 Percent	eon25Ansd	5.43 Percent
eon26Ansd	4.20 Percent	eon27Ansd	4.30 Percent	eon28Ansd	16.84 Percent
eon28Bnsd	17.44 Percent	eon29Ansd	15.88 Percent	eon30Ansd	16.04 Percent
eon30Bnsd	15.88 Percent	eon31Ansd	18.05 Percent	eon32Ansd	20.61 Percent
eon32Bnsd	20.11 Percent	eon33Ansd	23.59 Percent	eon33Bnsd	23.33 Percent
eon34Ansd	16.01 Percent	eon34Bnsd	16.00 Percent	eon21A3nsd	33.75 Percent
eon21B3nsd	29.30 Percent	eon22Cnsd	2.31 Percent	eon22Dnsd	2.41 Percent
eon23Cnsd	2.05 Percent	eon23Dnsd	2.16 Percent	eon24Cnsd	2.26 Percent
eon24Dnsd	2.37 Percent	eon25Cnsd	2.37 Percent	eon25Dnsd	2.52 Percent

Second Example of Cluster Usage

Normal (parallel) jobs.

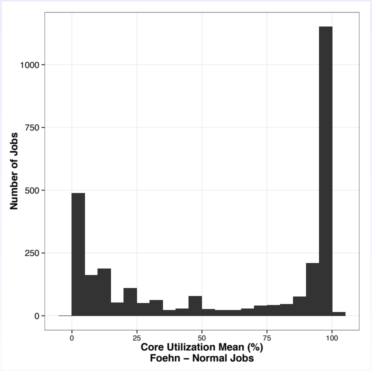


Fig. 1: Distribution of the jobs' core utilization means.

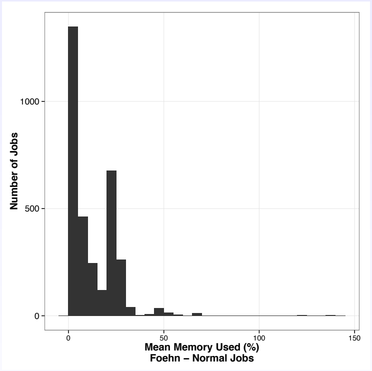


Fig. 2: Distribution of the mean memory used by job.

Second Example of Cluster Usage

Besteffort (mostly sequential) jobs.

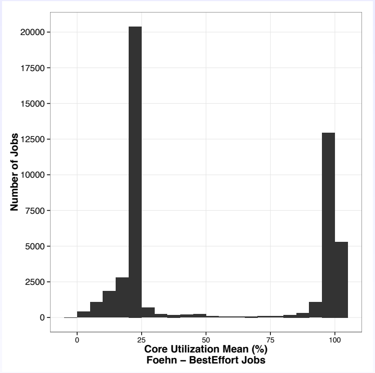


Fig. 5: Distribution of the jobs' core utilization means.

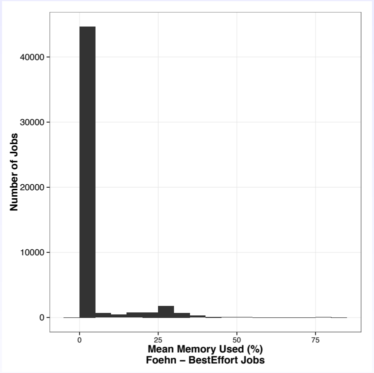


Fig. 6: Distribution of the mean memory used by job.

User reserves whole socket (4-cores) to run a memory-bandwidth greedy single-core job (using $\leq$ 620MB)

Inefficiency

- Space sharing with rigid allocations:
  - “Holes” in the schedule (unused nodes)
  - Unused resources within allocations (unused processors or cores in used nodes)
  - Missing tool to describe what is an acceptable sharing
- Integral allocations:
  - Idle resources (e.g., a task that needs only 20% of the CPU leaves it 80% idle while running)
  - Existing work focuses primarily on node utilization, NOT resource utilization
    Study of the 5 clusters of the DAS-2 platform: 30% of jobs use “no memory”, 16% of jobs use 324 KB, 33% of jobs use 2.6 to 3 MB.
    (Average usage of the Grid 5000 platform in 2008: 65%)
- One solution: fractional allocations

Virtual machine technology

Multiple jobs on one host

- Multiple tasks on one node (time sharing)
- Resource sharing
- Performance isolation: the monitor imposes precise resource shares
- Low-overhead and accurate: $\approx 1\%$
- Sharing of fractional resources
  Example: 1 CPU with two cores, 3 VM instances, each receiving 33.3% of the CPU capacity
- Enables preemption and migration

The Proposal

- Use virtual machine technology.
- Define a run-time computable metric that captures notions of performance and fairness.
- Design algorithms or heuristics that allocate resources to jobs while explicitly trying to achieve high ratings by the metric.

Outline

1 Introduction

2 The Steady-State Case

3 Algorithms

4 Simulation Experiments

5 Online Non-Clairvoyant Problem

6 Extension to heterogeneous platforms

7 Heuristics

8 Error

9 Conclusion

Outline

1 Introduction

2 The Steady-State Case

3 Algorithms

4 Simulation Experiments

5 Online Non-Clairvoyant Problem

6 Extension to heterogeneous platforms

7 Heuristics

8 Error

Requirements, Needs, and Yield

- Tasks have rigid requirements and fluid needs
    - All tasks of a parallel job have the same requirements and needs
- For a task to be placed on a node there must be rigid resources available at least equal to its requirements
- A task can be allocated less of a fluid resource than its need, but performance is *linearly* degraded
- The yield of a job is a value between 0 and 1; tasks are allocated an amount of each fluid resource equal to the job yield multiplied by the need in that resource

Requirements, Needs, and Yield

- Tasks have rigid requirements and fluid needs
  - All tasks of a parallel job have the same requirements and needs
- For a task to be placed on a node there must be rigid resources available at least equal to its requirements
- A task can be allocated less of a fluid resource than its need, but performance is *linearly* degraded
- The yield of a job is a value between 0 and 1; tasks are allocated an amount of each fluid resource equal to the job yield multiplied by the need in that resource

The yield of a job gives its performance relative to if it were run on a dedicated system.

Assumptions

- Steady-state execution with infinite jobs
  - Makes the problem more tractable [Marchal et al., 2006]
  - Avoids user estimates of job duration (notoriously inaccurate)
  - Good when job duration longer than schedule time
- Requirements and needs are constant throughout the execution
- Known resource requirements and needs
  - (Techniques available for discovery [Jones et al., 2006a, Jones et al., 2006b])

Static Problem

Additional assumptions

- The platform is supposed to be homogeneous: all hosts have the same characteristics
- All jobs are sequential jobs
- No use of migration

Resources

- Each server provides d types of resources
- For each job i, r_{ij} denotes its resource need or requirement for resource type j (fraction between 0 and 1)
- δ_{ij} is a binary value: 1 if r_{ij} is a requirement (rigid need), and 0 if r_{ij} is only a (fluid) need.

Objective Metric Considerations

- Performance and fairness
- One popular metric: *stretch* (or slowdown)
  - Time the job spends in the system divided by the time that would be spent in a dedicated system
  - Popular to quantify schedule quality post-mortem
  - Not used to make scheduling decisions
  - Minimizing the maximum stretch captures notions of both performance and fairness.
- Runtime computation requires user estimates
- Not applicable to infinite jobs

Yield

- Instantaneous version of the stretch
- Goal: Maximize the minimum yield
- Minimum yield can be “seen” as the inverse of the maximum stretch

Complexity

Maximizing the minimum yield

- Problem is NP-hard in the strong sense
- Even if no memory constraints
- Reduction from 3-Partition

Going further: the scaled yield

- Each job is set a *minimum yield*: minimum quality of service that is acceptable by the user (e.g., interactive applications, maximum response time)
- The minimum yield could be null
- The *scaled yield* of a service:

$$\text{scaled yield} = \frac{\text{yield} - \text{minimum yield}}{1 - \text{minimum yield}} \; .$$

- Objective: maximize the minimum scaled yield

Mixed Integer Linear Program 1/2

- Binary variable e_{ih}: whether task i runs on server h:

$$\forall i, h \quad e_{ih} \in \{0, 1\}$$

Mixed Integer Linear Program 1/2

- Binary variable e_{ih}: whether task i runs on server h:

$$\forall i, h \quad e_{ih} \in \{0, 1\}$$

- y_{ih} the (unscaled) yield of service i on server h

$$0 \leq y_{ih} \leq 1 \qquad y_{ih} \in \mathbb{Q}$$

Mixed Integer Linear Program 1/2

- Binary variable e_{ih}: whether task i runs on server h:

$$\forall i, h \quad e_{ih} \in \{0, 1\}$$

- y_{ih} the (unscaled) yield of service i on server h

$$0 \leq y_{ih} \leq 1 \qquad y_{ih} \in \mathbb{Q}$$

- Each job is executed exactly once

$$\forall i \quad \sum_{h} e_{ih} = 1$$

Mixed Integer Linear Program 1/2

- Binary variable e_{ih}: whether task *i* runs on server *h*:

$$\forall i, h \quad e_{ih} \in \{0, 1\}$$

- y_{ih} the (unscaled) yield of service *i* on server *h*

$$0 \leq y_{ih} \leq 1 \qquad y_{ih} \in \mathbb{Q}$$

- Each job is executed exactly once

$$\forall i \quad \sum_{h} e_{ih} = 1$$

- Yield not null on a host $\Rightarrow$ host executing the job

$$\forall i, h \quad 0 \leq y_{ih} \leq e_{ih}$$

Mixed Integer Linear Program 2/2

- The yield must at least be equal to the minimum yield

$$\forall i \quad \sum_{h} y_{ih} \geq \hat{y}_i$$

Mixed Integer Linear Program 2/2

- The yield must at least be equal to the minimum yield

$$\forall i \quad \sum_h y_{ih} \geq \hat{y}_i$$

- Resources must not be exceeded

$$\forall h, j \quad \sum_i r_{ij}(y_{ih}(1 - \delta_{ij}) + e_{ih}\delta_{ij}) \leq 1$$

Mixed Integer Linear Program 2/2

- The yield must at least be equal to the minimum yield

$$\forall i \quad \sum_h y_{ih} \geq \hat{y}_i$$

- Resources must not be exceeded

$$\forall h, j \quad \sum_i r_{ij}(y_{ih}(1 - \delta_{ij}) + e_{ih}\delta_{ij}) \leq 1$$

- Constraint on the minimum *scaled* yield

$$\forall i \quad \sum_h y_{ih} \geq \hat{y}_i + Y(1 - \hat{y}_i)$$

Mixed Integer Linear Program 2/2

- The yield must at least be equal to the minimum yield

$$\forall i \quad \sum_h y_{ih} \geq \hat{y}_i$$

- Resources must not be exceeded

$$\forall h, j \quad \sum_i r_{ij}(y_{ih}(1 - \delta_{ij}) + e_{ih}\delta_{ij}) \leq 1$$

- Constraint on the minimum *scaled* yield

$$\forall i \quad \sum_h y_{ih} \geq \hat{y}_i + Y(1 - \hat{y}_i)$$

- Objective: maximize the minimum scaled yield Y

Mixed Integer Linear Program: summary

Maximize Y under the constraints:

$$\begin{array}{rcr}
\forall i, h & e_{ih} \in \{0, 1\}\ , \quad y_{ih} \in \mathbb{Q} & (1) \\
\forall i & \sum_h e_{ih} = 1 & (2) \\
\forall i, h & 0 \leq y_{ih} \leq e_{ih} & (3) \\
\forall i & \sum_h y_{ih} \geq \hat{y}_i & (4) \\
\forall h, j & \sum_i r_{ij}(y_{ih}(1 - \delta_{ij}) + e_{ih}\delta_{ij}) \leq 1 & (5) \\
\forall i & \sum_h y_{ih} \geq \hat{y}_i + Y(1 - \hat{y}_i) & (6)
\end{array}$$

Outline

1 Introduction

2 The Steady-State Case

3 Algorithms

4 Simulation Experiments

5 Online Non-Clairvoyant Problem

6 Extension to heterogeneous platforms

7 Heuristics

8 Error

Solving the Problem

- We have defined an optimization problem, found that it's NP-hard, and produced an MILP formulation.
- *Question*: How do we solve it?
- We consider three classes of algorithms:
  - Exact solutions and absolute performance bounds
  - LP-based heuristics
  - Greedy heuristics
  - Multi-capacity bin packing heuristics

Exact and Relaxed Solutions

- MILP solving NP-hard
  - GNU Linear Programming Kit (GLPK) for small instances
- Rational (non-Integer) e_{ij}:
  - Solution in polynomial time
  - Generally not feasible
  - Produces upper bound on optimal
  - Useful to quantify performance of heuristics: absolute reference for the performance of heuristics

Linear Program-based heuristics: the bound

If the rational LP can be solved (i.e., the aggregate resource capacities can meet all rigid and constrained fluid needs), then it has an immediate solution:

$$Y = \min\left(1, \min_{j \in NZ} \frac{H - \sum_i r_{ij}(\hat{y}_i(1-\delta_{ij}) + \delta_{ij})}{\sum_i (1-\hat{y}_i) r_{ij}(1-\delta_{ij})}\right),$$

where NZ is the set of indices $j \in \{1, \ldots, d\}$ such that $\sum_i (1-\hat{y}_i) r_{ij}(1-\delta_{ij})$ is non-zero. This maximum minimum yield is achieved by the trivial allocation $e_{ih} = 1/H$ and $y_{ih} = \frac{1}{H}(\hat{y}_i + Y(1-\hat{y}_i))$, for all i and h.

Linear Program-based heuristics: rounding

- How to obtain an integer solution from a rational one?

Linear Program-based heuristics: rounding

- How to obtain an integer solution from a rational one?
- Randomized Rounding (RRND)

Linear Program-based heuristics: rounding

- How to obtain an integer solution from a rational one?
- Randomized Rounding (RRND)
  - Consider jobs in arbitrary order

Linear Program-based heuristics: rounding

- How to obtain an integer solution from a rational one?
- Randomized Rounding (RRND)
  - Consider jobs in arbitrary order
  - Assign job i to host h with probability e_{ih}

Linear Program-based heuristics: rounding

- How to obtain an integer solution from a rational one?
- Randomized Rounding (RRND)
  - Consider jobs in arbitrary order
  - Assign job i to host h with probability e_{ih}
  - If host h cannot accomodate job i sets e_{ih} to 0, scale non null e_{ij}'s and go back to previous step

Linear Program-based heuristics: rounding

- How to obtain an integer solution from a rational one?
- Randomized Rounding (RRND)
  - Consider jobs in arbitrary order
  - Assign job i to host h with probability e_{ih}
  - If host h cannot accomodate job i sets e_{ih} to 0, scale non null e_{ij}'s and go back to previous step
  - If no host can accomodate job i, the heuristic fails

Linear Program-based heuristics: rounding

- How to obtain an integer solution from a rational one?
- Randomized Rounding (RRND)
  - Consider jobs in arbitrary order
  - Assign job i to host h with probability e_{ih}
  - If host h cannot accomodate job i sets e_{ih} to 0, scale non null e_{ij}'s and go back to previous step
  - If no host can accomodate job i, the heuristic fails
- Randomized Rounding with No Zero probability (RRNDNZ)
  Same as previously except that all e_{ih} initially null are set to some value $\epsilon << 1$ ($\epsilon = 0.01$)

Linear Program-based heuristics: rounding

- How to obtain an integer solution from a rational one?
- Randomized Rounding (RRND)
  - Consider jobs in arbitrary order
  - Assign job i to host h with probability e_{ih}
  - If host h cannot accomodate job i sets e_{ih} to 0, scale non null e_{ij}'s and go back to previous step
  - If no host can accomodate job i, the heuristic fails
- Randomized Rounding with No Zero probability (RRNDNZ) Same as previously except that all e_{ih} initially null are set to some value $\epsilon << 1$ ($\epsilon = 0.01$)
- Iterative resolutions

Linear Program-based heuristics: rounding

- How to obtain an integer solution from a rational one?
- Randomized Rounding (RRND)
  - Consider jobs in arbitrary order
  - Assign job i to host h with probability e_{ih}
  - If host h cannot accomodate job i sets e_{ih} to 0, scale non null e_{ij}'s and go back to previous step
  - If no host can accomodate job i, the heuristic fails
- Randomized Rounding with No Zero probability (RRNDNZ) Same as previously except that all e_{ih} initially null are set to some value $\epsilon << 1$ ($\epsilon = 0.01$)
- Iterative resolutions
  - SLOWDIVING: Fix the value of the e_{ih} variable whose value is closest to its nearest rounding; then solve again the linear program

Linear Program-based heuristics: rounding

- How to obtain an integer solution from a rational one?
- Randomized Rounding (RRND)
  - Consider jobs in arbitrary order
  - Assign job i to host h with probability e_{ih}
  - If host h cannot accomodate job i sets e_{ih} to 0, scale non null e_{ij}'s and go back to previous step
  - If no host can accomodate job i, the heuristic fails
- Randomized Rounding with No Zero probability (RRNDNZ) Same as previously except that all e_{ih} initially null are set to some value $\epsilon << 1$ ($\epsilon = 0.01$)
- Iterative resolutions
  - SLOWDIVING: Fix the value of the e_{ih} variable whose value is closest to its nearest rounding; then solve again the linear program
  - FASTDIVING: assign the job i with the largest value e_{ih}; solve again the linear program

Greedy heuristics 1/4

Job sorting criteria

1. randomly;
2. sorted by decreasing maximum need;
3. by decreasing sum of needs;
4. by decreasing maximum requirement and constrained need;
5. by decreasing sum of requirements and constrained needs;
6. by decreasing maximum resource requirement or need;
7. by decreasing sum of requirements and needs.

Greedy heuristics 2/4

For a given job i

- Let j_n be the index of the resource corresponding to the maximum fluid need of i
- Let j_r be the index of the resource corresponding to the maximum requirement or constrained need of i
- Let I_h be the set of the indices of the jobs already placed on server h.

Greedy heuristics 3/4

Host sorting criteria to pick a server for a given job *i*

1. pick server *h* with the smallest $\max_{i' \in I_h} r_{i' j_n}$;
2. pick server *h* with the smallest $\sum_{i' \in I_h} r_{i' j_n}$.
3. Best fit approach evaluating the load of each server *h* based on $\max_{i' \in I_h} r_{i' j_r}$
4. Best fit approach evaluating the load of each server *h* based on $\sum_{i' \in I_h} r_{i' j_r}$
5. Worst-fit approach corresponding to 3
6. Worst-fit approach corresponding to 4
7. First fit placement, placing a job on the first server that can accommodate its requirements and constrained needs

Greedy heuristics 3/4

- $7 \times 7 = 49$ greedy heuristics
- These heuristics are lightweight
- Greedy heuristic number 50: run the 49 heuristics and takes the best solution (if any)

Vector Packing

- Problem similar to bin packing, 2 major differences
- Multiple dimensions: multi-dimensional bin-packing or *vector packing* or multi-capacity bin-packing
- Needs are only upper bounds not tight constraints
  Fix a tentative yield value
  - Needs become requirements
  - Can apply any vector-packing algorithm

  Binary search on the best achievable minimum yield

Best Fit and First Fit Vector Packing

- Best Fit and First Fit algorithm
- Sort vectors by
    - decreasing SUM of coordinates
    - Decreasing MAXimum of the coordinates
    - Decreasing LEXicographical order of the coordintes

All together: 6 algorithms

Permutation Pack and Choose Pack Vector Packing

- Motivation: attempt to balance the load along the dimensions
- PP partition the N vectors in one of the $d!/(d-w)!$ lists
  - Each list contains vectors with a common permutation of their largest w dimensions
  - Vectors in a list are sorted according to: decreasing SUM of coordinates, decreasing DIFFerence of largest to smallest coordinate, decreasing RATIO of largest to smallest coordinate
  - PP fills bin trying to reduce load imbalance: consider the w least loaded resource dimensions
- CP: relaxation of PP: Does not enforce any ordering between the w dimensions
  "Only" needs $d!/w!(w-d)!$ lists

Outline

1 Introduction

2 The Steady-State Case

3 Algorithms

4 Simulation Experiments

5 Online Non-Clairvoyant Problem

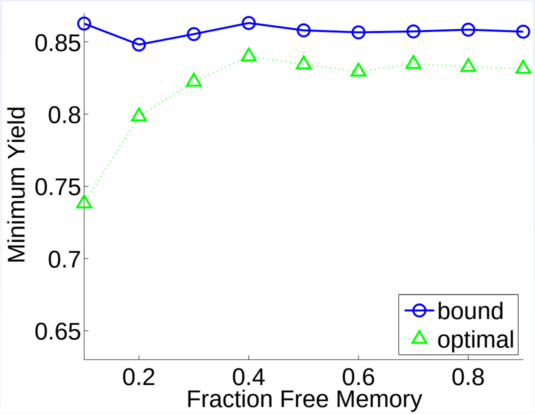
6 Extension to heterogeneous platforms

7 Heuristics

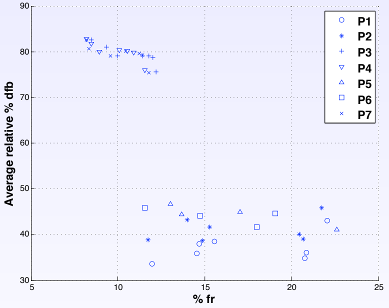
8 Error

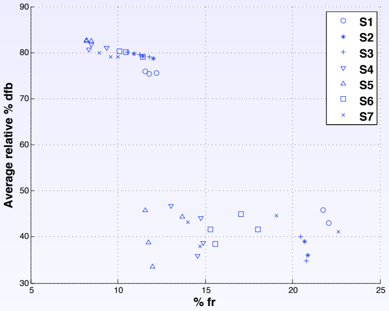
Small Problem Set: Minimum Yield vs. Free Memory

- Bound only 4% higher than optimal



Greedy heuristics





Bi-criteria graphical comparison of all GREEDY_Sx_Py algorithms, averaged over all 72,900 instances.

LP-based algorithms 1/2

Average *dfb*, 90th percentile *dfb*, and *fr*, for the LP-based algorithms and GREEDY, over 48,600 problem instances. Relative *dfb* values are shown in parentheses.

	dfb		
Algorithm	Average	90th perc.	*fr* (%)
RRND	0.58 (78.33%)	0.86 (98.52%)	66.56
RRNDNZ	0.58 (77.95%)	0.89 (98.28%)	22.02
FASTDIVING	0.60 (75.03%)	0.84 (94.39%)	78.02
SLOWDIVING	0.57 (72.75%)	0.81 (93.60%)	77.92
GREEDY	0.21 (29.17%)	0.38 (51.49%)	7.50

LP-based algorithms 2/2

Average execution times ($H = 64$ and $N = 128, 256, 512$.)

	Average Execution Time (sec)		
Algorithm	$N = 128$	$N = 256$	$N = 512$
RRND	16.30	61.70	255.44
RRNDNZ	16.74	61.15	250.83
FASTDIVING	32.42	113.89	416.32
SLOWDIVING	382.58	1771.35	6704.79
GREEDY	0.05	0.10	0.24

Vector-packing algorithms

	dfb				
Algorithm	Average		90th perc.		*fr* (%)
GREEDYLIGHT	0.16	(31.49%)	0.35	(56.25%)	8.16
GREEDY	0.16	(30.07%)	0.34	(61.01%)	7.73
VP_PPRATIO	0.08	(14.54%)	0.17	(28.32%)	15.81
VP_PPDIFF	0.08	(13.67%)	0.16	(21.10%)	15.35
VP_FFLEX	0.07	(12.85%)	0.15	(27.86%)	15.45
VP_PPMAX	0.07	(13.08%)	0.15	(26.67%)	14.99
VP_PPSUM	0.07	(12.84%)	0.15	(26.39%)	14.93
VP_CPRATIO	0.07	(11.09%)	0.14	(21.21%)	11.45
VP_BFLEX	0.06	(12.15%)	0.14	(27.10%)	13.75
VP_CPDIFF	0.06	(10.19%)	0.12	(21.10%)	8.70
VP_CPMAX	0.05	(10.10%)	0.11	(20.60%)	8.43
VP_CPSUM	0.05	(9.92%)	0.11	(20.40%)	8.20
VP_BFMAX	0.04	(11.39%)	0.11	(29.40%)	8.48
VP_FFMAX	0.04	(11.26%)	0.11	(29.33%)	8.33
VP_BFSUM	0.04	(10.95%)	0.10	(28.72%)	7.91
VP_FFSUM	0.04	(10.95%)	0.10	(28.40%)	7.91

Choose Pack (CP) dominates Permutation Pack (PP)

Vector-packing algorithms

	dfb				
Algorithm	Average		90th perc.		*fr* (%)
GREEDYLIGHT	0.16	(31.49%)	0.35	(56.25%)	8.16
GREEDY	0.16	(30.07%)	0.34	(61.01%)	7.73
VP_PPRATIO	0.08	(14.54%)	0.17	(28.32%)	15.81
VP_PPDIFF	0.08	(13.67%)	0.16	(21.10%)	15.35
VP_FFLEX	0.07	(12.85%)	0.15	(27.86%)	15.45
VP_PPMAX	0.07	(13.08%)	0.15	(26.67%)	14.99
VP_PPSUM	0.07	(12.84%)	0.15	(26.39%)	14.93
VP_CPRATIO	0.07	(11.09%)	0.14	(21.21%)	11.45
VP_BFLEX	0.06	(12.15%)	0.14	(27.10%)	13.75
VP_CPDIFF	0.06	(10.19%)	0.12	(21.10%)	8.70
VP_CPMAX	0.05	(10.10%)	0.11	(20.60%)	8.43
VP_CPSUM	0.05	(9.92%)	0.11	(20.40%)	8.20
VP_BFMAX	0.04	(11.39%)	0.11	(29.40%)	8.48
VP_FFMAX	0.04	(11.26%)	0.11	(29.33%)	8.33
VP_BFSUM	0.04	(10.95%)	0.10	(28.72%)	7.91
VP_FFSUM	0.04	(10.95%)	0.10	(28.40%)	7.91

VP_CPDIFF dominates VP_CPRATIO

Vector-packing algorithms

	dfb		
Algorithm	Average	90th perc.	*fr* (%)
GREEDYLIGHT	0.16 (31.49%)	0.35 (56.25%)	8.16
GREEDY	0.16 (30.07%)	0.34 (61.01%)	7.73
VP_PPRATIO	0.08 (14.54%)	0.17 (28.32%)	15.81
VP_PPDIFF	0.08 (13.67%)	0.16 (21.10%)	15.35
VP_FFLEX	0.07 (12.85%)	0.15 (27.86%)	15.45
VP_PPMAX	0.07 (13.08%)	0.15 (26.67%)	14.99
VP_PPSUM	0.07 (12.84%)	0.15 (26.39%)	14.93
VP_CPRATIO	0.07 (11.09%)	0.14 (21.21%)	11.45
VP_BFLEX	0.06 (12.15%)	0.14 (27.10%)	13.75
VP_CPDIFF	0.06 (10.19%)	0.12 (21.10%)	8.70
VP_CPMAX	0.05 (10.10%)	0.11 (20.60%)	8.43
VP_CPSUM	0.05 (9.92%)	0.11 (20.40%)	8.20
VP_BFMAX	0.04 (11.39%)	0.11 (29.40%)	8.48
VP_FFMAX	0.04 (11.26%)	0.11 (29.33%)	8.33
VP_BFSUM	0.04 (10.95%)	0.10 (28.72%)	7.91
VP_FFSUM	0.04 (10.95%)	0.10 (28.40%)	7.91

VP_CPSUM is never dominated

Vector-packing algorithms

Algorithm	*dfb* Average		*dfb* 90th perc.		*fr* (%)
GREEDYLIGHT	0.16	(31.49%)	0.35	(56.25%)	8.16
GREEDY	0.16	(30.07%)	0.34	(61.01%)	7.73
VP_PPRATIO	0.08	(14.54%)	0.17	(28.32%)	15.81
VP_PPDIFF	0.08	(13.67%)	0.16	(21.10%)	15.35
VP_FFLEX	0.07	(12.85%)	0.15	(27.86%)	15.45
VP_PPMAX	0.07	(13.08%)	0.15	(26.67%)	14.99
VP_PPSUM	0.07	(12.84%)	0.15	(26.39%)	14.93
VP_CPRATIO	0.07	(11.09%)	0.14	(21.21%)	11.45
VP_BFLEX	0.06	(12.15%)	0.14	(27.10%)	13.75
VP_CPDIFF	0.06	(10.19%)	0.12	(21.10%)	8.70
VP_CPMAX	0.05	(10.10%)	0.11	(20.60%)	8.43
VP_CPSUM	0.05	(9.92%)	0.11	(20.40%)	8.20
VP_BFMAX	0.04	(11.39%)	0.11	(29.40%)	8.48
VP_FFMAX	0.04	(11.26%)	0.11	(29.33%)	8.33
VP_BFSUM	0.04	(10.95%)	0.10	(28.72%)	7.91
VP_FFSUM	0.04	(10.95%)	0.10	(28.40%)	7.91

Lexicographical ordering algorithms are dominated

Vector-packing algorithms

Algorithm	*dfb* Average		*dfb* 90th perc.		*fr* (%)
GreedyLight	0.16	(31.49%)	0.35	(56.25%)	8.16
Greedy	0.16	(30.07%)	0.34	(61.01%)	7.73
VP_PPRatio	0.08	(14.54%)	0.17	(28.32%)	15.81
VP_PPDiff	0.08	(13.67%)	0.16	(21.10%)	15.35
VP_FFLex	0.07	(12.85%)	0.15	(27.86%)	15.45
VP_PPMax	0.07	(13.08%)	0.15	(26.67%)	14.99
VP_PPSum	0.07	(12.84%)	0.15	(26.39%)	14.93
VP_CPRatio	0.07	(11.09%)	0.14	(21.21%)	11.45
VP_BFLex	0.06	(12.15%)	0.14	(27.10%)	13.75
VP_CPDiff	0.06	(10.19%)	0.12	(21.10%)	8.70
VP_CPMax	0.05	(10.10%)	0.11	(20.60%)	8.43
VP_CPSum	0.05	(9.92%)	0.11	(20.40%)	8.20
VP_BFMax	0.04	(11.39%)	0.11	(29.40%)	8.48
VP_FFMax	0.04	(11.26%)	0.11	(29.33%)	8.33
VP_BFSum	0.04	(10.95%)	0.10	(28.72%)	7.91
VP_FFSum	0.04	(10.95%)	0.10	(28.40%)	7.91

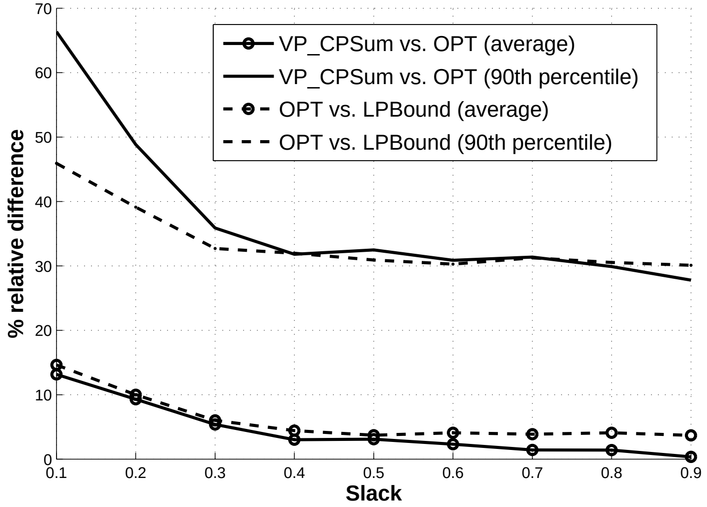
VP_FFSum is as good or better than VP_FFMax

Vector-packing algorithms

	dfb				
Algorithm	Average		90th perc.		*fr* (%)
GREEDYLIGHT	0.16	(31.49%)	0.35	(56.25%)	8.16
GREEDY	0.16	(30.07%)	0.34	(61.01%)	7.73
VP_PPRATIO	0.08	(14.54%)	0.17	(28.32%)	15.81
VP_PPDIFF	0.08	(13.67%)	0.16	(21.10%)	15.35
VP_FFLEX	0.07	(12.85%)	0.15	(27.86%)	15.45
VP_PPMAX	0.07	(13.08%)	0.15	(26.67%)	14.99
VP_PPSUM	0.07	(12.84%)	0.15	(26.39%)	14.93
VP_CPRATIO	0.07	(11.09%)	0.14	(21.21%)	11.45
VP_BFLEX	0.06	(12.15%)	0.14	(27.10%)	13.75
VP_CPDIFF	0.06	(10.19%)	0.12	(21.10%)	8.70
VP_CPMAX	0.05	(10.10%)	0.11	(20.60%)	8.43
VP_CPSUM	0.05	(9.92%)	0.11	(20.40%)	8.20
VP_BFMAX	0.04	(11.39%)	0.11	(29.40%)	8.48
VP_FFMAX	0.04	(11.26%)	0.11	(29.33%)	8.33
VP_BFSUM	0.04	(10.95%)	0.10	(28.72%)	7.91
VP_FFSUM	0.04	(10.95%)	0.10	(28.40%)	7.91

VP_BFSUM is as good or better than VP_BFMAX

Comparison to optimal



Summary

- We have formalized a new cluster resource allocation approach that implicitly relies on virtualization technology to:
  - Increase cluster utilization through fractional resource sharing
  - Optimize a precise metric that captures notions of performance and fairness
- Algorithm of Choice: VP_CPSUM

Outline

1 Introduction

2 The Steady-State Case

3 Algorithms

4 Simulation Experiments

5 Online Non-Clairvoyant Problem

6 Extension to heterogeneous platforms

7 Heuristics

8 Error

HPC Scheduling Problem Overview

- We assume a cluster of homogeneous nodes dedicated to processing user jobs
- Users can submit requests at arbitrary times
- Running jobs are made up of nearly identical tasks
  - The number of tasks is specified by the user
  - Tasks may need to block while communicating
- Jobs are temporary
  - May have to wait until resources are available to start
  - Runtime is not known in advance

Stretch

- Stretch: the time a job spends in the system divided by the time that would be spent in a dedicated system [Bender et al., 1998]
- Popular to quantify schedule quality post-mortem
- Not generally used to make scheduling decisions
- Runtime computation requires (unreliable) user estimates
- Minimizing average stretch prone to starvation
- Minimizing maximum stretch captures notions of both performance and fairness [Legrand et al., 2008]

Optimal Lower Bound

- Given a clairvoyant scenario and infinite system memory, can compute a max-stretch lower bound in P-time
- Bound may not be achievable in practice
- Useful for comparing the performance of scheduling algorithms

Our Approach

- Basic idea: Consider an on-line maximum stretch minimization problem instance as a sequence of off-line minimum yield maximization problem instances
- Need heuristics to map tasks to nodes
- Need additional heuristics to allocate resources
- Need to decide when to apply heuristics

Task Placement Heuristics

We apply task placement heuristics studied for the off-line problem [**?**]:

- **Greedy Task Placement** – Incremental, puts each task on the node with the lowest computational load on which it can fit without violating memory constraints
- **MCB Task Placement** – Global, iteratively applies multi-capacity (vector) bin-packing heuristics during a binary search for the maximized minimum yield
  - Achieves higher minimum yield values than Greedy
  - Can *potentially* cause lots of migration
- But what if the system is oversubscribed?
  - Need a priority function to decide which jobs to run

Virtual Time

Definition

The virtual time $v_j(t)$ of job j at time t is the subjective time experienced by the job.

- $v_j(t) = \int_{r_j}^{t} y_j(\tau) d\tau$
- job completes when $v_j(t) =$ execution time (not known beforehand)

The Need for Preemption

- final goal is to minimize maximum stretch
- without preemption, stretch of non-clairvoyant on-line algorithms unbounded
  - consider 2 jobs
  - both require all of the system resources
  - one has $c_j = 1$
  - other has $c_j = \Delta$
- need criteria to decide which jobs should be preempted

Priority

Jobs should be preempted in order by increasing priority.

- Newly arrived jobs may have infinite priority
- First Idea: $\frac{1}{\text{VIRTUAL TIME}}$
  - Informed by ideas about fairness
  - Lead to good results
  - But theoretically prone to starvation
- Second Idea: $\frac{\text{FLOW TIME}}{\text{VIRTUAL TIME}}$
  - Addresses starvation problem
  - But lead to poor performance
- Third Idea: $\frac{\text{FLOW TIME}}{(\text{VIRTUAL TIME})^2}$
  - Combines idea #1 and idea #2
  - Addresses starvation
  - Performs about the same as first priority function

Use of Priority

- By Greedy
  - **GreedyP** – Greedily schedule tasks, and suspend lower-priority tasks if necessary to run higher-priority tasks
  - **GreedyPM** – Like **GreedyP**, but can also migrate tasks instead of suspending them
- By MCB
  - If no valid solution can be found for any yield value, remove the lowest priority task and try again

Resource Allocation

- Once tasks are placed on nodes we iteratively maximize the minimum yield
- Based on network resource allocation ideas about fairness
- Easy to compute and slightly better than maximizing average yield

When to apply Heuristics

We consider a number of different options:

- Job Submission – heuristics can use greedy or bin packing approaches
- Job Completion – as above, can help with throughput when there are lots of short running jobs
- Periodically – some heuristics periodically apply vector packing to improve overall job placement

MCB-Stretch Algorithm

- Like MCB, but tries to minimize maximum stretch
- Requires knowledge of time until next rescheduling period, uses current and estimated future stretch
- Second phase focuses on iteratively minimizing the maximum stretch

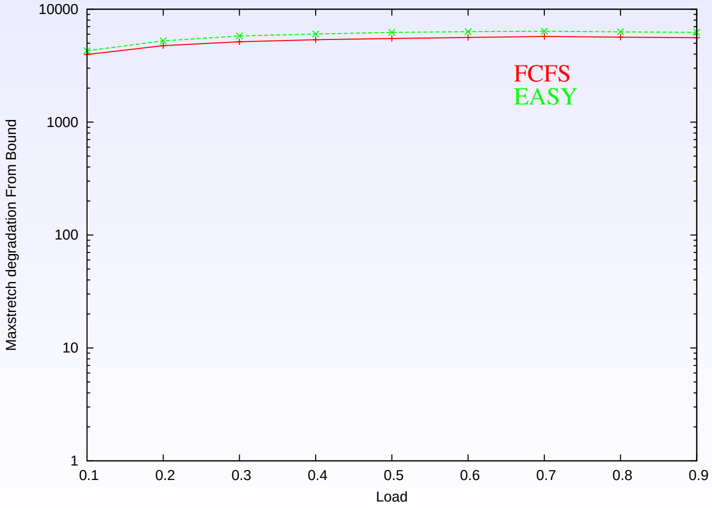
Methodology

- Experiments conducted using discrete event simulator
- Mix of synthetic and real trace data
- Ran experiments with and without migration penalties
- Periodic approaches use a 600 second (10 minute) period
- Absolute bound on max stretch computed for each instance
- Performance comparison based on max stretch degradation from bound

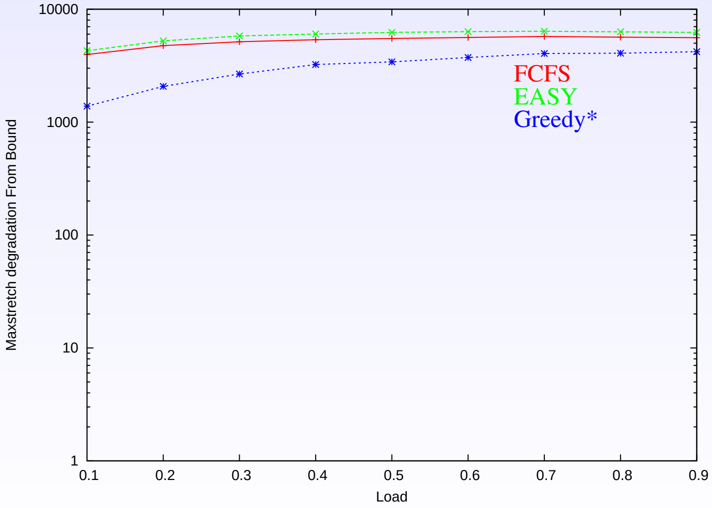
Batch Scheduling Algorithms

- FCFS – Allocates nodes equal to the number of tasks to jobs on a first-come-first-served basis.
- EASY – Only makes a reservation for the first job in the queue. Otherwise allocates nodes to the first job in the queue that can run with the current number of available nodes. Requires (unreliable) user-supplied run-time estimates to make reservations.

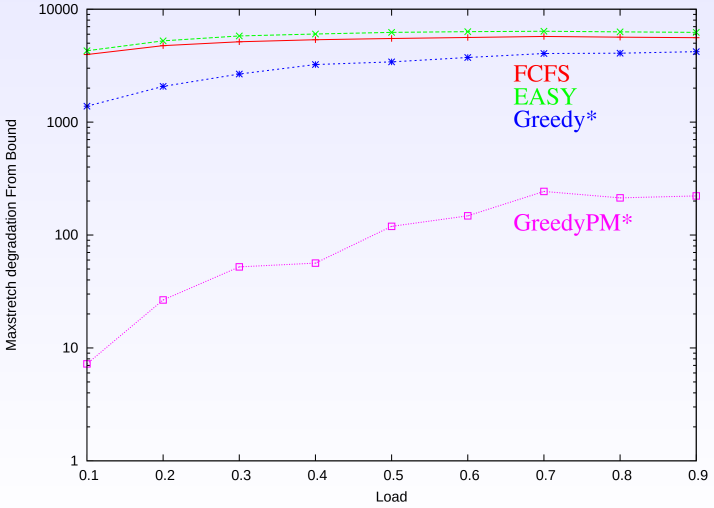
Max Stretch Degradation vs. Load, No Migration Cost



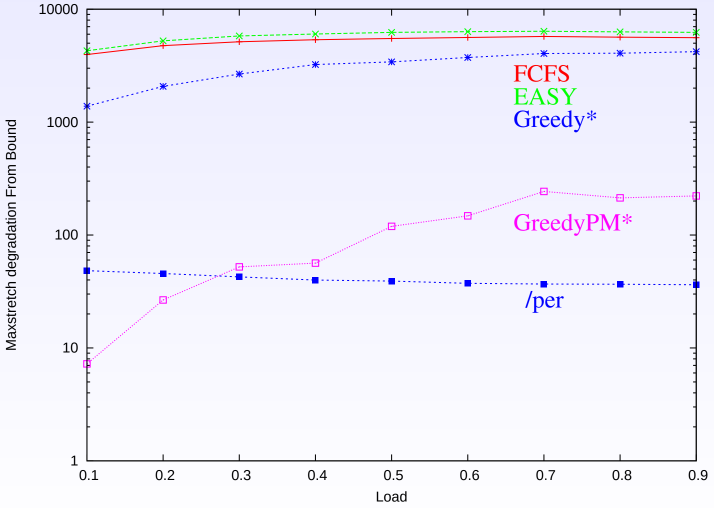
Max Stretch Degradation vs. Load, No Migration Cost



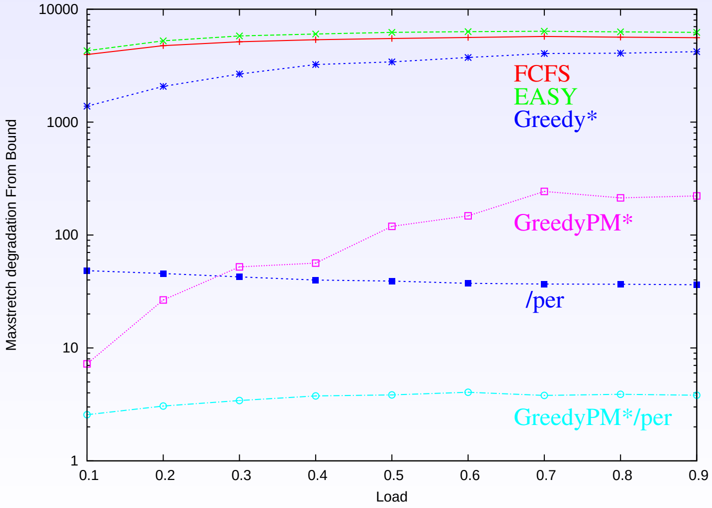
Max Stretch Degradation vs. Load, No Migration Cost



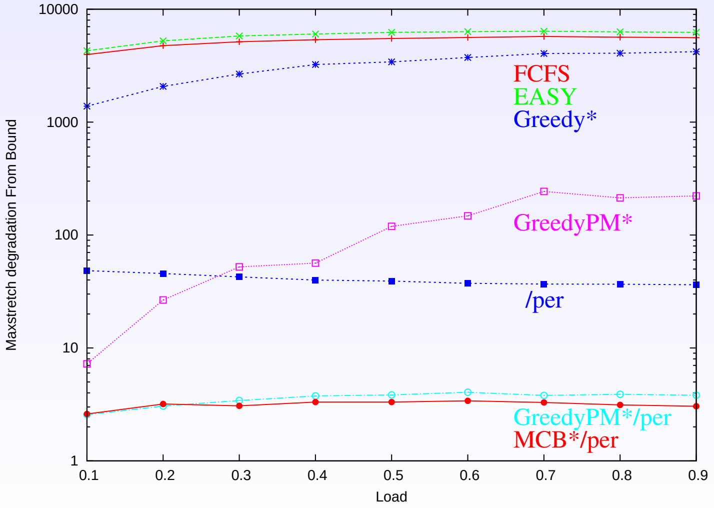
Max Stretch Degradation vs. Load, No Migration Cost



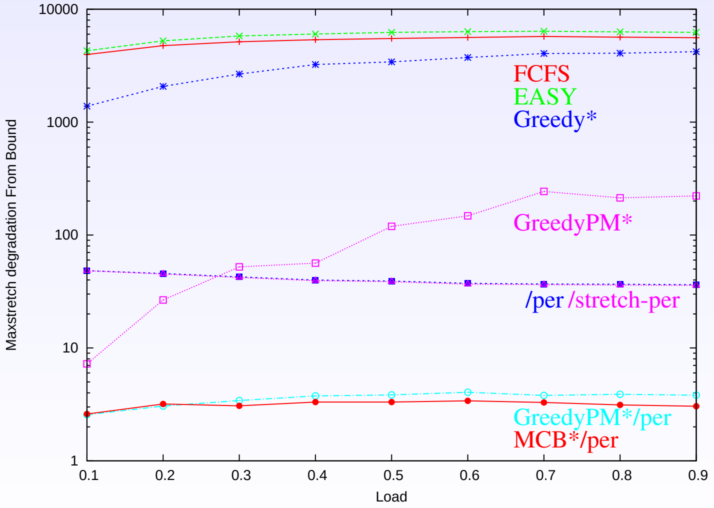
Max Stretch Degradation vs. Load, No Migration Cost



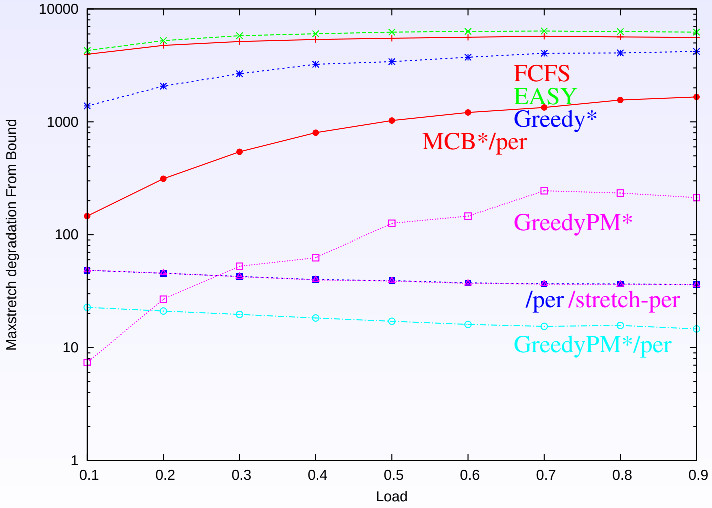
Max Stretch Degradation vs. Load, No Migration Cost



Max Stretch Degradation vs. Load, No Migration Cost



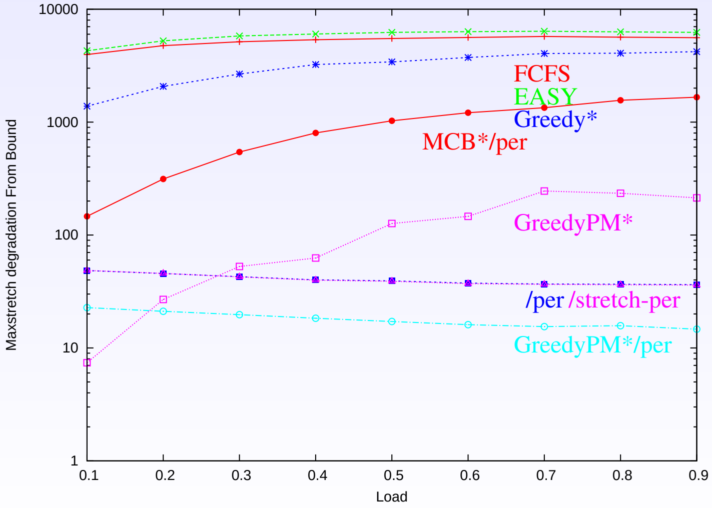
Max Stretch Degradation vs. Load, 5 minute penalty



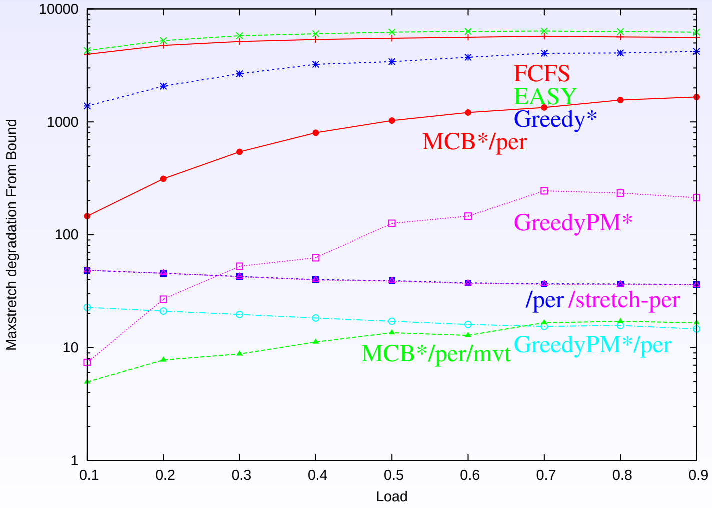
Limiting Migration

- Short-running jobs suffer a greater penalty to stretch from preemption/migration
- No way to tell short from long running jobs a-priori
- We do know the subjective time experienced by a job
- The minvt parameter specifies the minimum virtual time for a job before it can be migrated
- Does not affect preemption due to priority
- We tried 300 seconds and 600 seconds, 600 performed slightly better

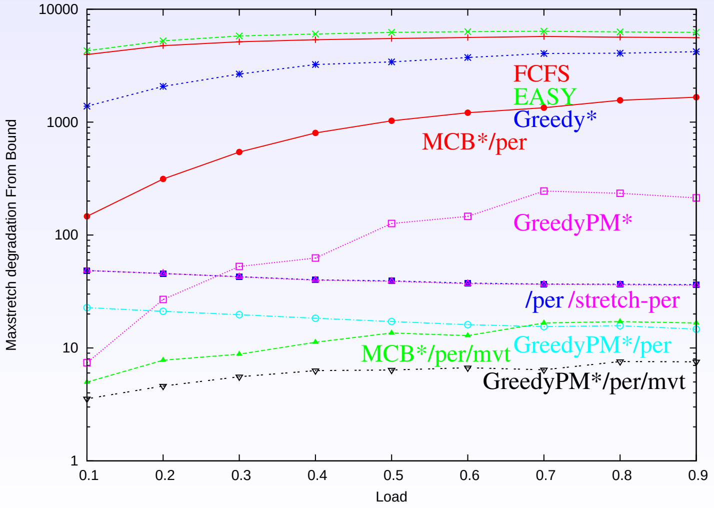
Max Stretch Degradation vs. Load, 5 minute penalty



Max Stretch Degradation vs. Load, 5 minute penalty



Max Stretch Degradation vs. Load, 5 minute penalty



Bandwidth Utilization

Preemption and migration bandwidth costs for selected algorithms. Average and maximum values over scaled synthetic traces with load $\geq$ 0.7

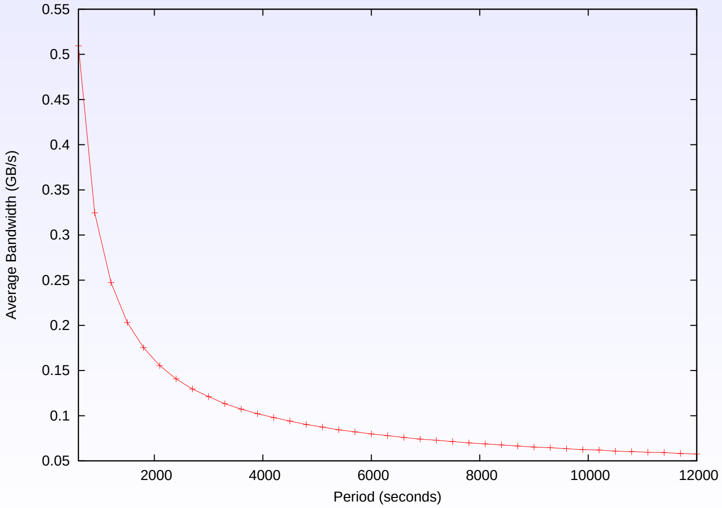
Algorithm	(GB / sec)			
	pmtn		mig	
	avg.	max	avg.	max
GreedyPM *	0.03	0.07	0.02	0.05
GreedyPM */per	0.56	1.37	0.29	0.66
GreedyPM */per/MVT	0.54	1.34	0.26	0.62
MCB */per/MVT	0.54	1.11	0.56	1.53
/per/MVT	0.49	1.08	0.19	0.58
/stretch-per/MVT	0.28	0.64	0.37	0.78

Bandwidth Utilization

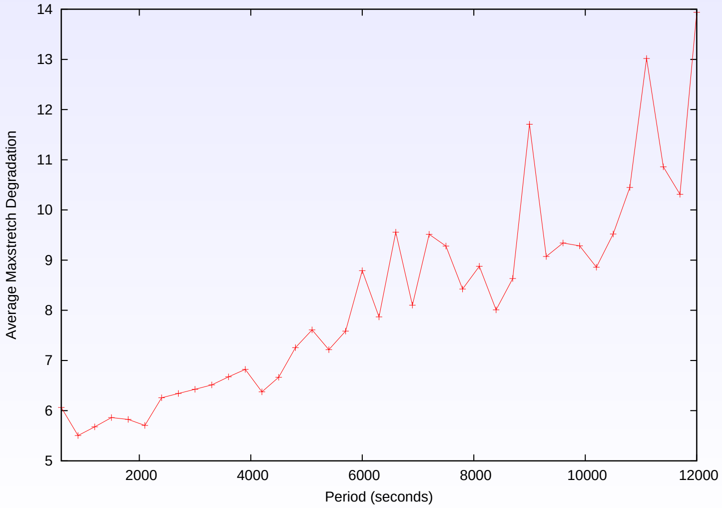
Preemption and migration bandwidth costs for selected algorithms. Average and maximum values over scaled synthetic traces with load ≥ 0.7

Algorithm	(GB / sec)			
	pmtn		mig	
	avg.	max	avg.	max
GreedyPM *	0.03	0.07	0.02	0.05
GreedyPM */per	0.56	1.37	0.29	0.66
GreedyPM */per/MVT	0.54	1.34	0.26	0.62
MCB */per/MVT	0.54	1.11	0.56	1.53
/per/MVT	0.49	1.08	0.19	0.58
/stretch-per/MVT	0.28	0.64	0.37	0.78

Bandwidth vs. Period



Max Stretch Degradation vs. Period



Conclusions

- DFRS algorithms are capable of widely outperforming traditional approaches, even assuming a heavy penalty for migration
- A variety of approaches need to be combined in order to achieve the best results
- Bandwidth costs are reasonable, and can be further reduced without a significant performance penalty by choosing an appropriate rescheduling period from a broad range

Summary

- We have proposed a novel approach to job scheduling on clusters, Dynamic Fractional Resource Scheduling, that makes use of modern virtual machine technology and seeks to optimize a runtime-computable, user-centric measure of performance called the minimum yield
- Our approach avoids the use of unreliable runtime estimates
- This approach has the potential to lead to order-of-magnitude improvements in performance over current technology
- Overhead costs from migration are manageable

Outline

1 Introduction

2 The Steady-State Case

3 Algorithms

4 Simulation Experiments

5 Online Non-Clairvoyant Problem

6 Extension to heterogeneous platforms

7 Heuristics

8 Error

Heterogeneous Environments

- federated cloud services
- system maintenance and upgrades
- repurposed hardware / budget shops
- volunteer computing

Elementary vs. Aggregate Capacity

- numerous types of resources
- easy to aggregate:
  - memory
  - disk space
- more problematic (time shared...):
  - processor
  - network / disk I/O
- for each node resource an elementary and aggregate capacity
  - aggregate usually an integer multiple of elementary
  - for easy resources they are the same

Service Hosting Jobs

- one VM per job
- infinite / very long time horizon
    - can reschedule / reshuffle periodically

Requirements vs. Needs

- rigid fixed requirements and fluid performance-bound needs
  - at both the elementary and aggregate levels
- classically, “CPU needs” and “memory requirements”
  - minimum processing rate $\rightarrow$ CPU requirement
  - scalable memory cache $\rightarrow$ memory need

Yield

- $0 < \text{yield} \leq 1$
- allocation $=$ requirement $+$ yield $\times$ need
- goal: find a mapping and allocation that maximizes the minimum yield
    - focus on performance AND fairness
    - maximizing average yield leads to starvation

Finding Solutions

- NP-complete problem
- can be formulated as MILP
  - for small problems solvers (GLPK, CPLEX) can provide optimal solutions
  - not practical for large problems
- instead we focus on heuristics

Outline

1 Introduction

2 The Steady-State Case

3 Algorithms

4 Simulation Experiments

5 Online Non-Clairvoyant Problem

6 Extension to heterogeneous platforms

7 Heuristics

8 Error

Types of Heuristics

- LP-based
- Greedy
- Vector Packing

RRND/RRNZ

- relax MILP to LP (fractional placements)
- round randomly by weight
  - disallow invalid allocations
- no zero probabilities improves chance of success

Greedy

- place each job on the “best” currently available host
- different criteria for sorting jobs
- different criteria for selecting hosts
- all algorithms are very fast
- METAGREEDY: try them all and use the best

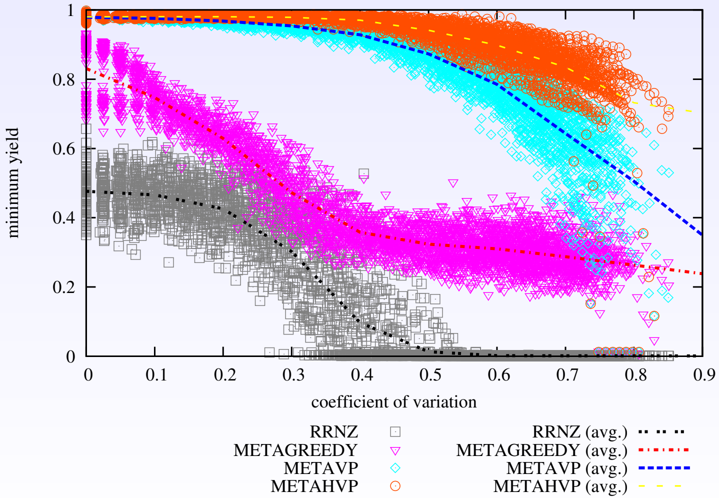
Vector Packing

- for a given yield, apply a vector packing heuristic
  - First-Fit, Best Fit, Choose Pack, Permutation Pack...
- binary search to find the highest yield instance solved
- VP heuristics also very fast, so METAVP
- VP heuristics can be extended to heterogeneous bins
  - H. First-Fit, H. Best-Fit, H. Choose Pack...
  - METAHVP

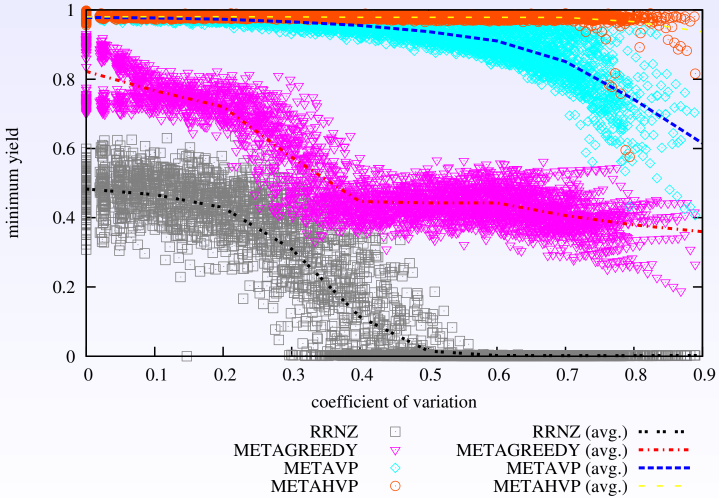
Experiments

- synthetic workloads
- 64 nodes
- 100, 250, 500 tasks
- memory slack 0.1-0.9
- CPU needs / memory requirements distribution based on Google cluster dataset
- node resource COV from 0.00 (homogeneous) to 1.00 (heterogeneous)

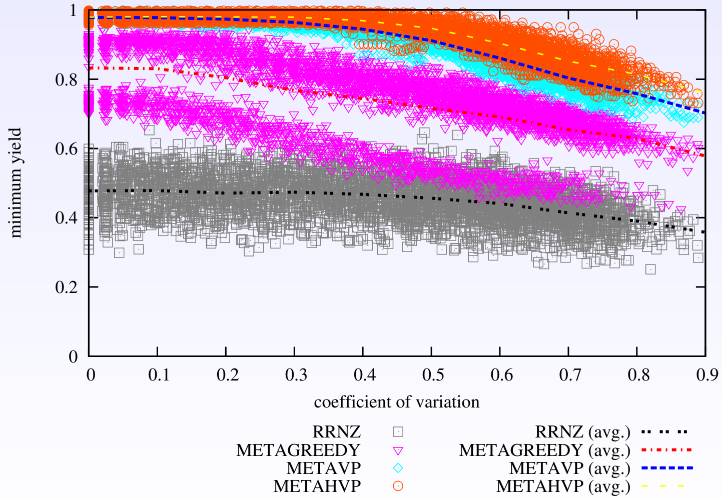
Performance vs. COV



Homogeneous Memory



Homogeneous CPU



Outline

1 Introduction

2 The Steady-State Case

3 Algorithms

4 Simulation Experiments

5 Online Non-Clairvoyant Problem

6 Extension to heterogeneous platforms

7 Heuristics

8 Error

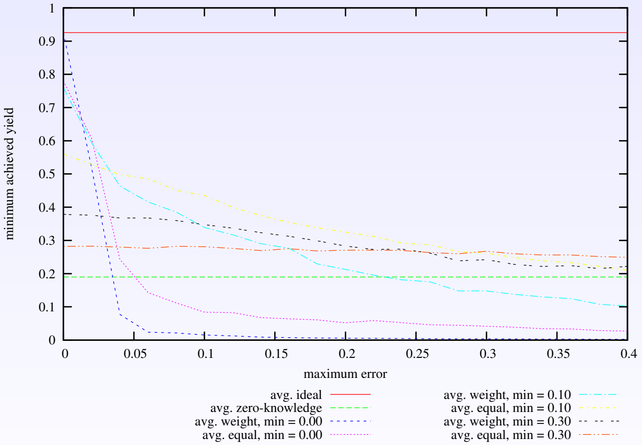
Needs Estimate Errors

- some requirements (e.g., memory) are relatively easy to determine and characterize
- others, like CPU, are more difficult
  - no accepted models
  - changes over time, sometimes quickly
  - difficult to define precisely

Placement and Allocation Strategies

- Zero-Knowledge (single-node $\frac{2J-1}{J^2}$ competitive ratio)
- Using Estimates:
    - assign caps
    - assign weights
    - equal access on each node

Consequences of Error



Achieved Minimum Yield from Baseline vs Maximum Specified Error on problem instances with 64 hosts, 250 services, memory slack = 0.4, coefficient of variation = 0.5

Outline

1 Introduction

2 The Steady-State Case

3 Algorithms

4 Simulation Experiments

5 Online Non-Clairvoyant Problem

6 Extension to heterogeneous platforms

7 Heuristics

8 Error

Summary

- Resource Allocation Problem:
  - NP complete, expressed as MILP
  - heterogeneous resource capacities
  - node elementary / aggregate resources
  - task resource requirements and needs
  - goal: maximize minimum performance
- Heuristics:
  - Greedy
  - Vector Packing
  - Heterogeneous Vector Packing
- Errors in Resource Need Estimates
  - capping usage very wasteful
  - naïve approach worse than zero-knowledge
  - we can compensate (somewhat)

References I

Bender, M. A., Chakrabarti, S., and Muthukrishnan, S. (1998).
Flow and stretch metrics for scheduling continuous job streams.
In *SODA*, pages 270–279.

Jones, S. T., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. (2006a).
Antfarm: Tracking processes in a virtual machine environment.
In *USENIX*.

References II

Jones, S. T., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. (2006b).
Geiger: Monitoring the buffer cache in a virtual machine environment.
In *ASPLOS*.

Lee, C. B. and Snavely, A. E. (2007).
Precise and realistic utility functions for user-centric performance analysis of schedulers.
In *HPDC*, pages 107–116.

Legrand, A., Su, A., and Vivien, F. (2008).
Minimizing the stretch when scheduling flows of divisible requests.
J. Sched.
to appear, DOI: 10.1007/s10951-008-0078-4.

References III

Marchal, L., Yang, Y., Casanova, H., and Robert, Y. (2006).
Steady-state scheduling of multiple divisible load applications on wide-area distributed computing platforms.
J. HPCA, 20(3):365–381.

Mixed-Integer Linear Program

$$\begin{aligned}
\forall j, h \quad & e_{jh} \in \{0, 1\}, \quad y_{jh} \in [0, 1], \quad y_{jh} \leq e_{jh} \\
\forall j \quad & \sum_{h=1}^{H} e_{jh} = 1 \\
\forall j, h, d \quad & e_{jh} r^e_{jd} + y_{jh} n^e_{jd} \leq c^e_{hd} \\
\forall h, d \quad & \sum_{j=1}^{J} (e_{jh} r^a_{jd} + y_{jh} n^a_{jd}) \leq c^a_{hd} \\
\forall j \quad & \sum_{h=1}^{H} y_{jh} \geq Y
\end{aligned}$$

Run Times

Algorithm	100 tasks	250 tasks	500 tasks
RRNZ	4.855	45.782	270.245
METAGREEDY	0.014	0.061	0.154
METAVP	0.142	0.564	1.715
METAHVP	0.514	1.943	6.432

Average seconds on Intel Xeon 2.27Ghz processor

References I

Bender, M. A., Chakrabarti, S., and Muthukrishnan, S. (1998).
Flow and stretch metrics for scheduling continuous job streams.
In *SODA*, pages 270–279.

Jones, S. T., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. (2006a).
Antfarm: Tracking processes in a virtual machine environment.
In *USENIX*.

References II

Jones, S. T., Arpaci-Dusseau, A. C., and Arpaci-Dusseau, R. H. (2006b).
Geiger: Monitoring the buffer cache in a virtual machine environment.
In *ASPLOS*.

Lee, C. B. and Snavely, A. E. (2007).
Precise and realistic utility functions for user-centric performance analysis of schedulers.
In *HPDC*, pages 107–116.

Legrand, A., Su, A., and Vivien, F. (2008).
Minimizing the stretch when scheduling flows of divisible requests.
J. Sched.
to appear, DOI: 10.1007/s10951-008-0078-4.

References III

Marchal, L., Yang, Y., Casanova, H., and Robert, Y. (2006).
Steady-state scheduling of multiple divisible load applications on wide-area distributed computing platforms.
J. HPCA, 20(3):365–381.