Sharing Computational Resources on the Grid: How to do it Reliably and Unselfishly

Saurabh Bagchi
Dependable Computing Systems Lab (DCSL)
School of ECE, Purdue University

Joint work with: Tanzima Zerin, Rudi Eigenmann (Purdue)
Kathryn Mohror, Adam Moody, Bronis Supinski (LLNL)

Work supported by
Department of Energy,
National Science Foundation,
Purdue Research Foundation

Greetings come to you from …

What are Cycle Sharing Distributed Systems?

- Systems with following characteristics
  - Harvests idle cycles of Internet connected hosts
  - Enforces host owners' priority in utilizing resources
  - Resource becomes unavailable whenever owners are "active"

- Popular examples: Climateprediction.net and the World Community Grid
- Computing infrastructure provided by active open source codebases such as Boinc

What are Fine-Grained Cycle Sharing Systems?

- Cycle Sharing systems with following characteristics
  - Allows foreign jobs to coexist on a machine with local ("submitted by owner") jobs
  - Resource becomes unavailable if slowdown of local jobs is observable
  - Resource becomes unavailable if machine fails or is intentionally removed from the network

Fine-Grained Cycle Sharing: FGCS

  - Example: Condor

Trouble in “FGCS Land”

- Uncertainty of execution environment to remote jobs
- Result of fluctuating resource availability
  - Resource contention and revocation by machine owner
  - Software-hardware faults
  - Abrupt removal of machine from network
- Resource unavailability is not rare
  - More than 450 occurrences per machine in traces collected during 4 months on 20 machines

How to handle fluctuating resource availability?

Current state of practice

- Reactive Approach
  - Do nothing till the failure happens
  - Restart the job on a different machine in the cluster
- Proactive Approach
  - Predict when resource will become unavailable
  - Migrate job prior to failure and restart on different machine, possibly from checkpoint
- Advantage of proactive approach: Completion time of job is shorter

IF, prediction can be done accurately and efficiently

Our Contributions

1. Prediction of Resource Availability in FGCS
   - Multi-state availability model
     - Integrates general system failures with domain-specific resource behavior in FGCS
   - Prediction using a semi-Markov Process model
     - Accurate, fast, and robust
2. Integration of failure prediction in scheduler for production FGCS
   - Reduction in application completion time
3. Efficient checkpointing on shared grid resources
   - How to share checkpointing resources
   - How to optimize size of checkpoints stored

Outline

- Multi-State Availability Model
  - Different classes of unavailability
  - Methods to detect unavailability
- Prediction Algorithm
  - Semi-Markov Process model
- Evaluation Results
  - Computational cost
  - Prediction accuracy
  - Robustness to irregular history data
  - Results of Failure Prediction in a Scheduler
- Checkpointing

Two Types of Resource Unavailability

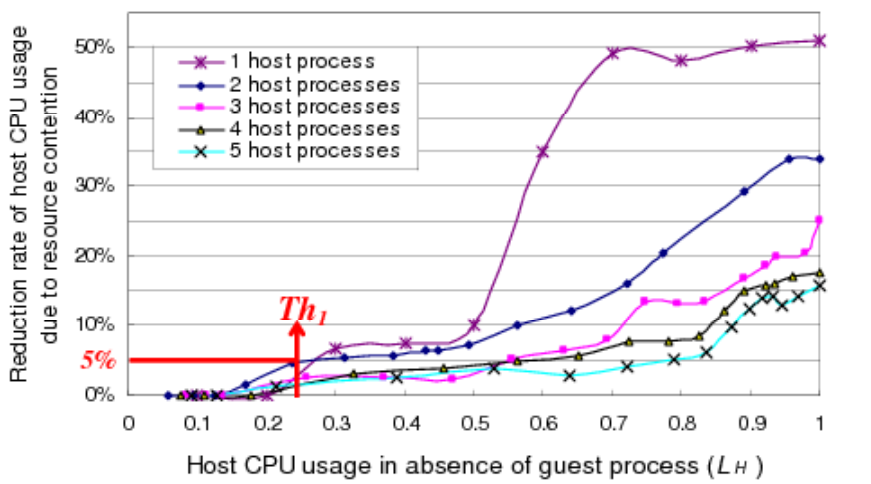
- **Target applications** – Long running, either CPU or memory intensive, batch mode
- ***UEC*** – Unavailability due to Excessive Resource Contention
  - Resource contention among one ***guest*** job and ***host*** jobs (CPU and memory)
  - Policy to handle resource contention: Host jobs are sacrosanct
    - Decrease the guest job's priority if host jobs incur *noticeable slowdown*
    - Terminate the guest job if slowdown still persists
- ***URR*** – Unavailability due to Resource Revocation
  - Machine owner's intentional leave
  - Software-hardware failures

Studies on Resource Contention

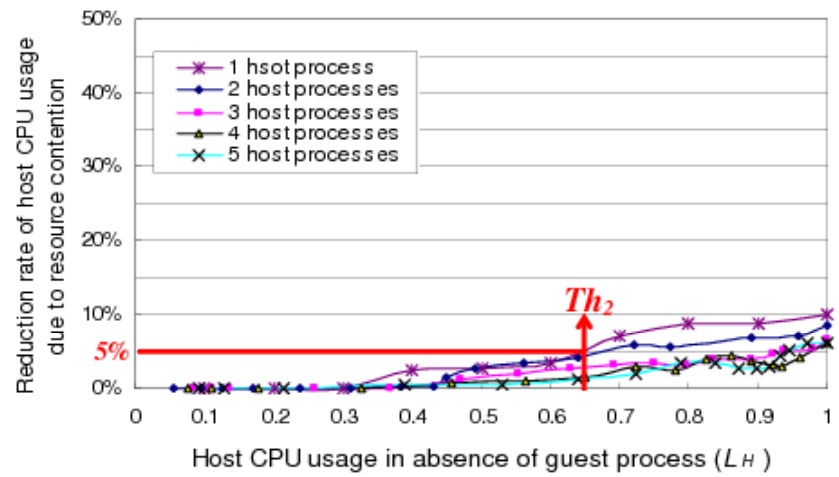
- Detecting UEC requires quantification of noticeable slowdown of host jobs
  - *Noticeable slowdown* of host jobs cannot be measured directly
- Our detection method
  - Threshold for acceptable slowdown in host CPU usage is 5%
  - Empirically find the correlation between observed machine CPU usage and effect on host job due to contention from the guest job

Studies on Resource Contention: CPU

- Experiment settings
  - CPU-intensive guest job
  - *Host group*: Multiple host jobs with different CPU usages
  - Measure CPU reduction of host group for different sizes of host group



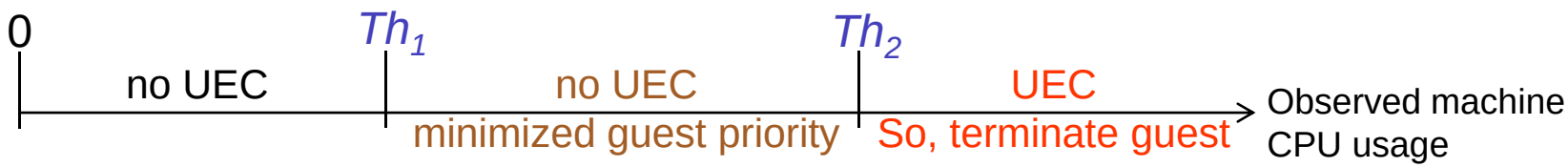
(a) All processes have the same priority



(b) Guest process takes the lowest priority

Studies on Resource Contention: CPU

- CPU contention study shows the existence of two thresholds Th_1 and Th_2 in machine CPU usage
- UEC can be detected by observing machine CPU usage on Linux systems

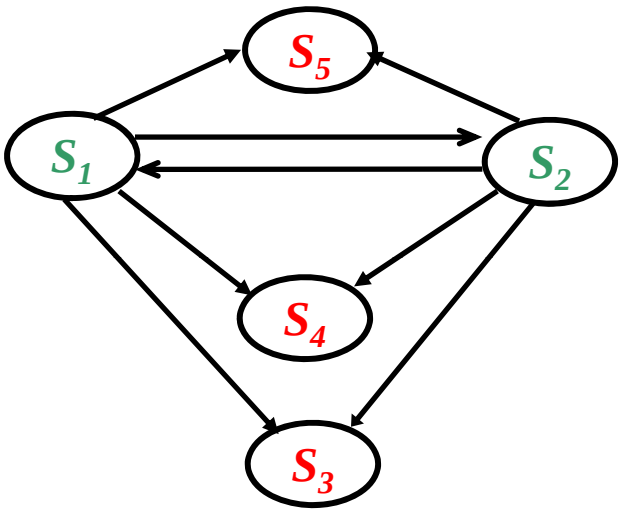


- Other candidate policies for regulating guest job
  - Vary priority with a finer granularity: Experiments show no significant difference for L_H in (20%, 50%)
  - Always run guest job with lowest priority: Too conservative and may incur unacceptable slowdown

Studies on Resource Contention: CPU & Memory

- Evaluate larger applications that have significant CPU and memory utilization
- Experiment settings
  - Guest applications: SPEC CPU2000 benchmark suite
  - Host workload: Musbus Unix benchmark suite
  - 300 MHz Solaris Unix machine with 464 MB physical memory
  - Measure host CPU reduction by running a guest application together with a set of host workloads
- Observations
  - Memory thrashing happens when jobs desire more memory than the system has
  - The memory thrashing persists even if guest job priority is reduced
  - When there is sufficient memory, UEC solely depends on the host CPU usage
  - Thus CPU and memory contention can be considered independent

Multi-State Resource Availability Model



S_1: Machine CPU load is $[0\%, Th_1]$

S_2: Machine CPU load is $(Th_1, Th_2]$

S_3: Machine CPU load is $(Th_2, 100\%]$ -- UEC

S_4: Memory thrashing -- UEC

S_5: Machine unavailability -- URR

For guest jobs, S_3, S_4, and S_5 are unrecoverable failure states

Resource Availability Prediction

- Goal of Prediction
  - Predict temporal reliability (TR)

    The probability that resource will be available throughout a future time window
- Semi-Markov Process (SMP)
  - States and transitions between states
  - Probability of transition to next state depends only on current state and amount of time spent in current state (independent of history)
- Algorithm for TR calculation:
  - Construct an SMP model from history data for the same time windows on previous days

    Daily patterns of host workloads are comparable among recent days
  - Compute TR for the predicted time window

Background on SMP

- Probabilistic Models for Analyzing Dynamic Systems

  S : state

  Q : transition probability matrix

  $Q_i(j)$ = Pr {the process that has entered S_i will enter S_j on its next transition};

  H : holding time mass function matrix

  $H_{i,j}(m)$ = Pr {the process that has entered S_i remains at S_i for m time units before the next transition to S_j}

- Interval Transition Probabilities, P

  $P_{i,j}(m) = \Pr\{S(t_0+m)=j \mid S(t_0)=i\}$

Solving Interval Transition Probabilities

- Continuous-time SMP
  - Forward Kolmogorov integral equations

$$P_{i,j}(m) = \sum_{k \in S} \int_0^m Q_i(k) \times H_{i,k}(u) \times P_{k,j}(m-u)\,du$$

Too inefficient for online prediction

- Discrete-time SMP
  - Recursive equations

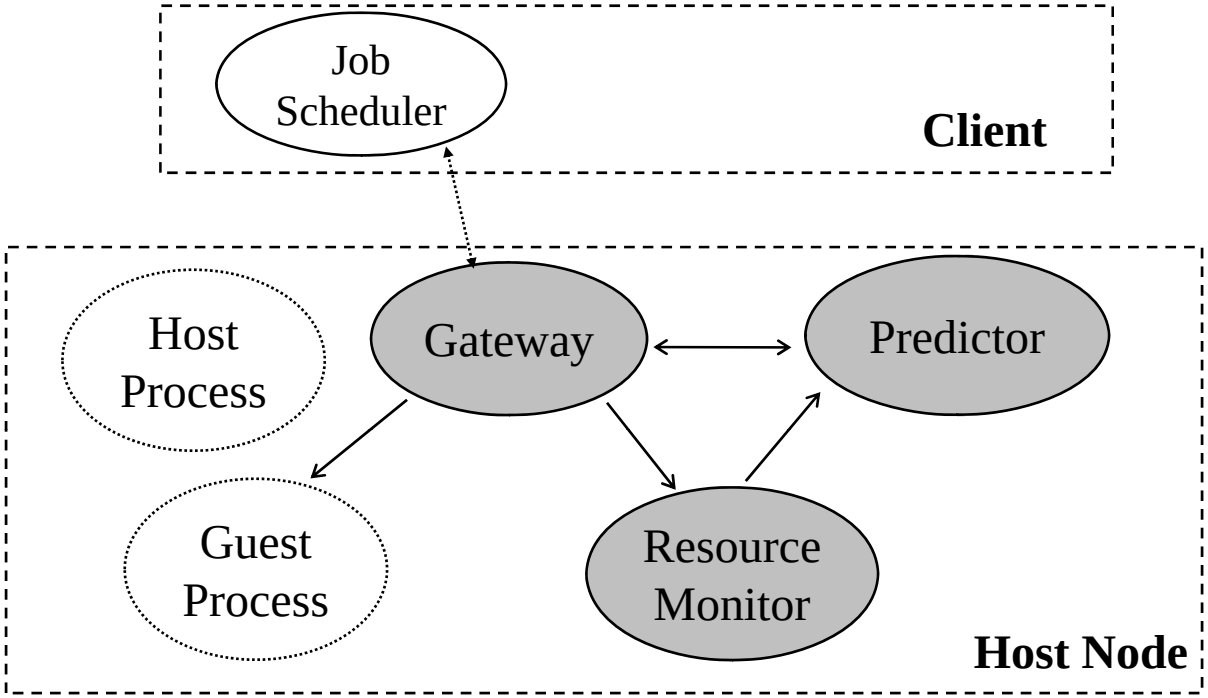
$$P_{i,j}(m) = \sum_{l=1}^{m-1} P_{i,k}^{1}(l) \times P_{k,j}(m-l) = \sum_{l=1}^{m-1} \sum_{k \in S} Q_i(k) \times H_{i,k}(l) \times P_{k,j}(m-l)$$

- Availability Prediction

TR(W): the probability of not transferring to S_3, S_4, or S_5 within an arbitrary time window, W of size T

$$TR(W) = 1 - [P_{init,3}(T/d) + P_{init,4}(T/d) + P_{init,5}(T/d)]$$

System Implementation



Non-intrusive monitoring of resource availability

- Use lightweight system utilities to measure CPU and memory load of host processes in non-privileged mode
- Example: `vmstat`, `prstat`

Evaluation of Availability Prediction

- Testbed
  - A collection of 1.7 GHz Redhat Linux machines in a student computer lab at Purdue
    - Reflect the multi-state availability model
    - Contain highly diverse host workloads
  - 1800 machine-days of traces measured in 3 months
- Statistics on Resource Unavailability

Categories	**Total amount**	**UEC**		**URR**
		CPU contention	Memory contention	
Frequency	405-453	283-356	83-121	3-12
Percentage	100%	69-79%	19-30%	0-3%

Evaluation Approach

- Metrics
  - Overhead: monitoring and prediction
  - Accuracy
  - Robustness
- Approach
  - Divide the collected trace into training and test data sets
  - Parameters of SMP are learnt based on training data
  - Evaluate the accuracy by comparing the prediction results for test data
  - Evaluate the robustness by inserting noise into training data set

Reference Algorithms: Linear Time Series Models

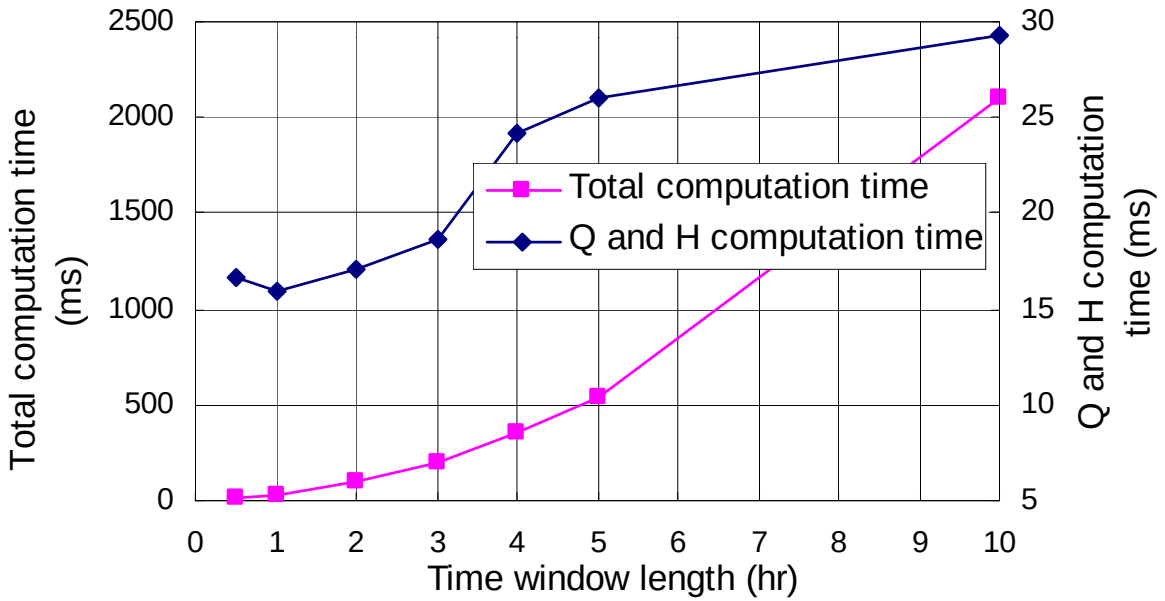
- Widely used for CPU load prediction in Grids: Network Weather Service*
- Linear regression equations**
- Application in our availability prediction
  - Predict future system states after observing training set
  - Compare the observed TR on the predicted and measured test sets

*R. Wolski, N. Spring, and J. Hayes, The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing, *JFGCS*, 1999

** Toolset from P. A. Dinda and D. R. O'Halaron. "An evaluation of linear models for host load prediction". In Proc. Of HPDC'99.

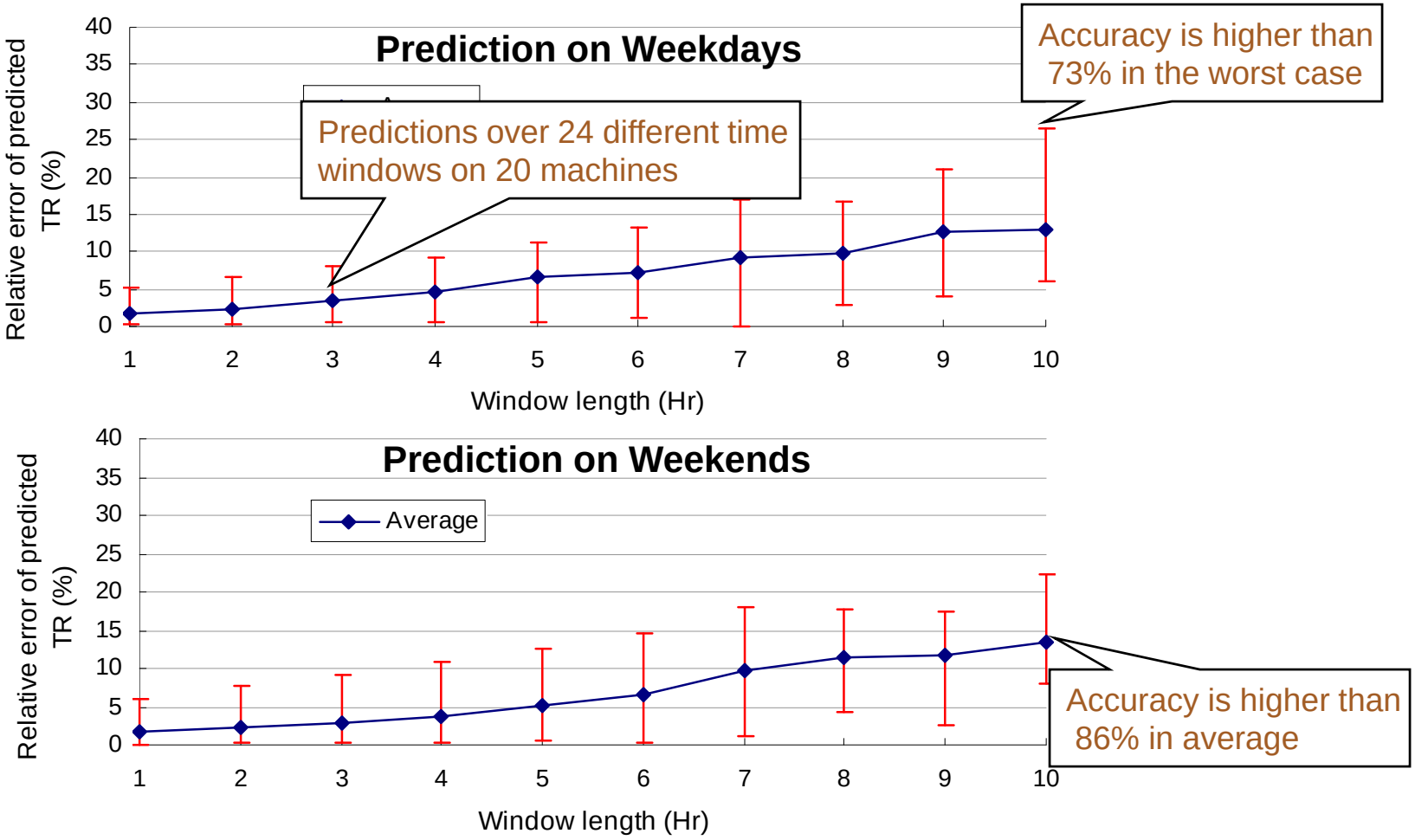
Overhead

- Resource Monitoring Overhead: CPU 1%, Memory 1%
- Prediction Overhead



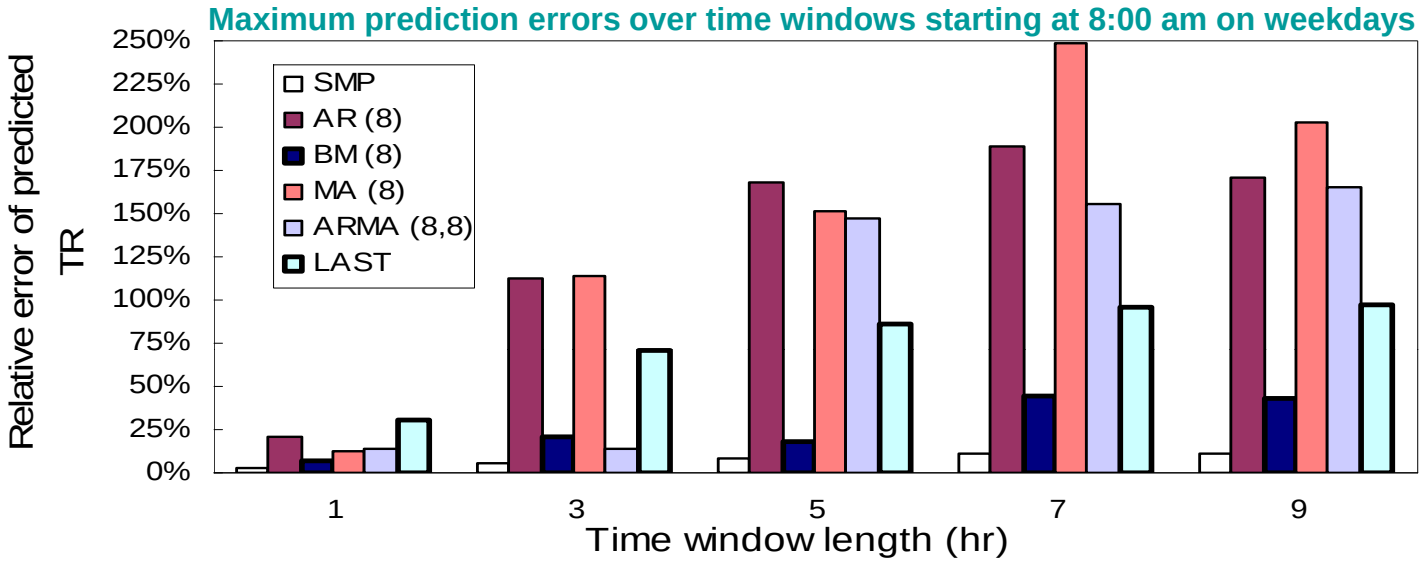
Less than 0.006% overhead to a remote job

Prediction Accuracy



Relative error = abs($TR_{predicted}$-$TR_{empirical}$)/$TR_{empirical}$

Comparison with Linear Time Series Models



Resource Prediction System: http://www.cs.cmu.edu/~cmcl/remulac/remos.html

Model	Description
AR(p)	Purely autoregressive models with p coefficients
BM(p)	Mean over the previous N values (N < p)
MA(p)	Moving average models with p coefficients
ARMA(p,q)	Autoregressive moving average models with p+q coefficients
LAST	Last measured values

Using Resource Prediction in Job Scheduling

- Compare three schedulers
  - Falcon which uses failure prediction
  - Condor which does matchmaking but is oblivious to failures
  - Idealized omniscient algorithm
- How to integrate failure prediction

Estimated Job Completion Time (JCT) without failures = $\frac{TL}{CR \times [1 - L_{t_0}(TL)]}$

MTTF = $\int_0^{\infty} TR(t)dt$

Effective Task Length (ETL) = MTTF $\times$ CR $\times$ (1-L_{t_0}(MTTF))

t_0 is job submission time, CR is the clock rate of the machine, $L_{t_0}(T)$ is the average load from (t_0, t_0+T).

Falcon Scheduler

1. Compare MTTF with JCT.
2. If JCT > MTTF for each machine, select the one with largest ETL.
3. If JCT < MTTF, select the one with minimum job completion time considering failures (JCTF)

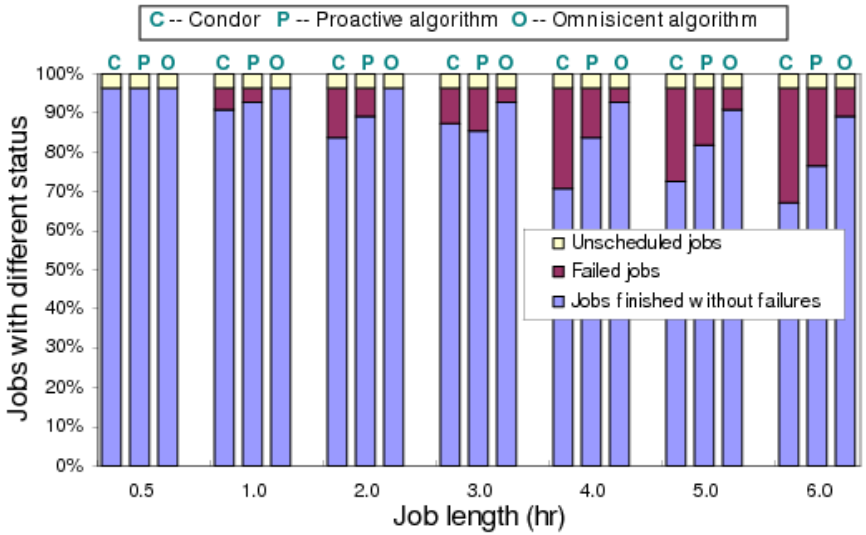
$$TL = CR \times (1 - L_{t_0}(JCTF)) \times \int_0^{JCTF} TR(t)dt$$

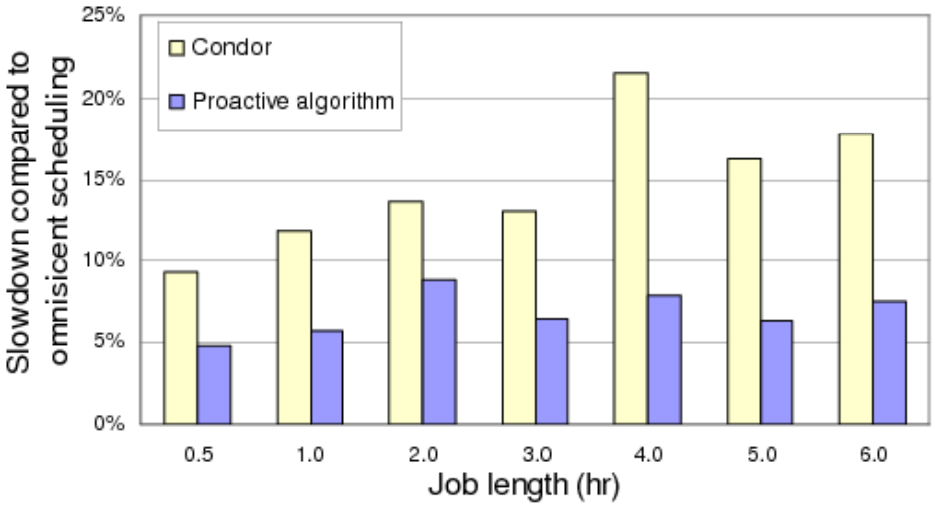
TL: Task length, the job completion time on a dedicated host machine with the average CPU speed as in our FGCS testbed

Experiments for Schedulers

- Experiments performed to compare three scheduling algorithms
  - Using GridSim but using availability traces from testbed
  - Jobs submitted with submission times between 6 am and 10 pm, 7 different lengths from 0.5 to 6 hours
- Experiment Metrics
  - When a job is submitted, if no resource is available, then wait for 5 minutes and retry; if still no resource available, then *unscheduled*
  - A scheduled job either returns as *finished*, or if resource becomes unavailable then *failed*
  - A failed job is restarted until it completes successfully

Results from Scheduler Experiments





- Fraction of unscheduled jobs same
- Difference between proactive and omniscient algorithms increase with job length
- For large jobs, fraction of failures lower for iShare than Condor
- Non monotonicity at 3 hours

- Omniscient algorithm has no overhead for prediction
- Two sources of slowdown – prediction overhead, ineffectiveness of prediction
- Effect of failure and restart adversely affects Condor
- Improvement in makespan 4-14%

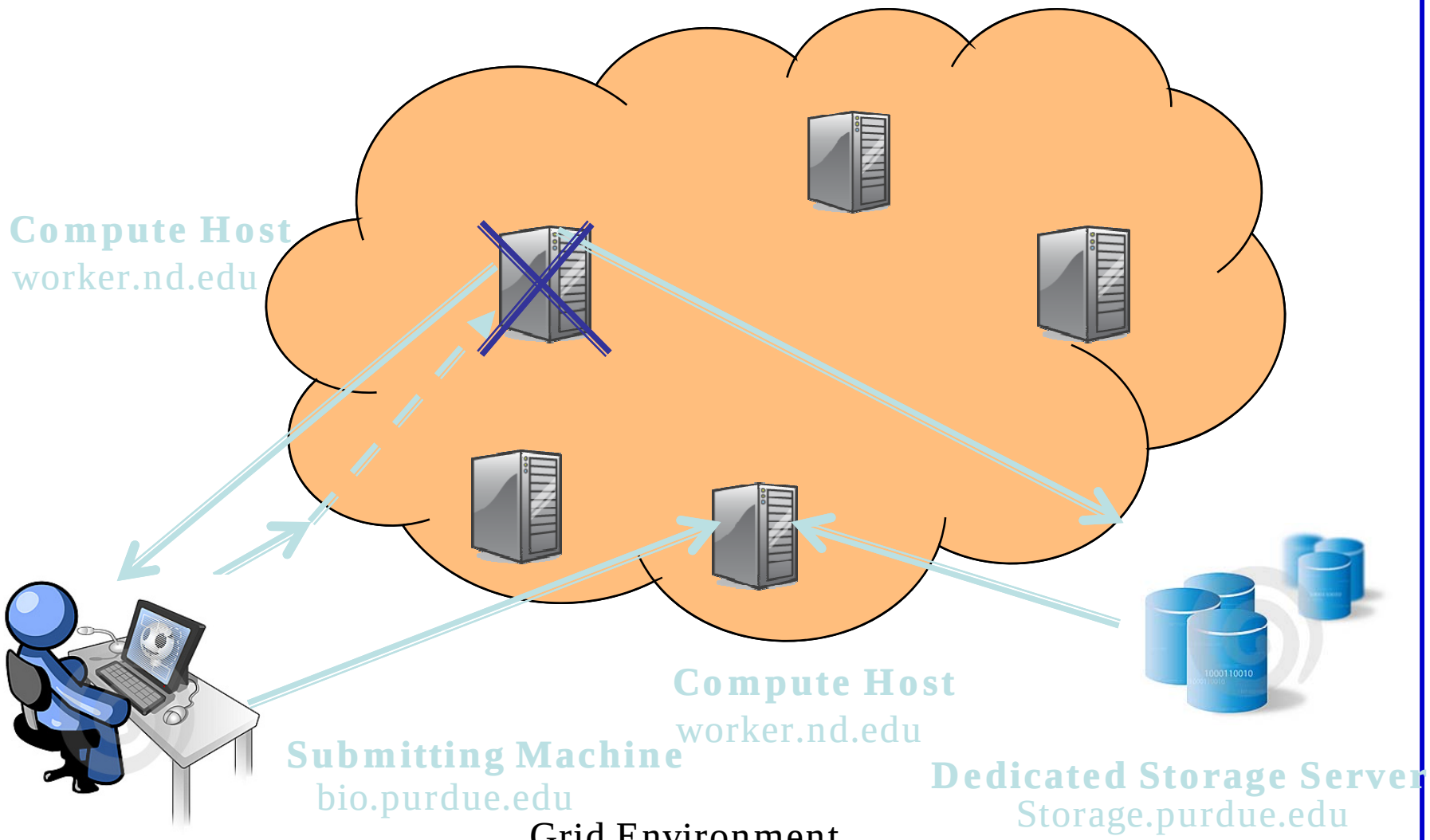
Checkpoint and Restart

- When a job is evicted from a machine, it needs to be restarted on another machine
- To avoid recomputation, the job should be restarted from a checkpoint

- Questions that we answer:

1. Where to save the checkpoints?
2. How to compress the checkpoints so that state stored is reduced?

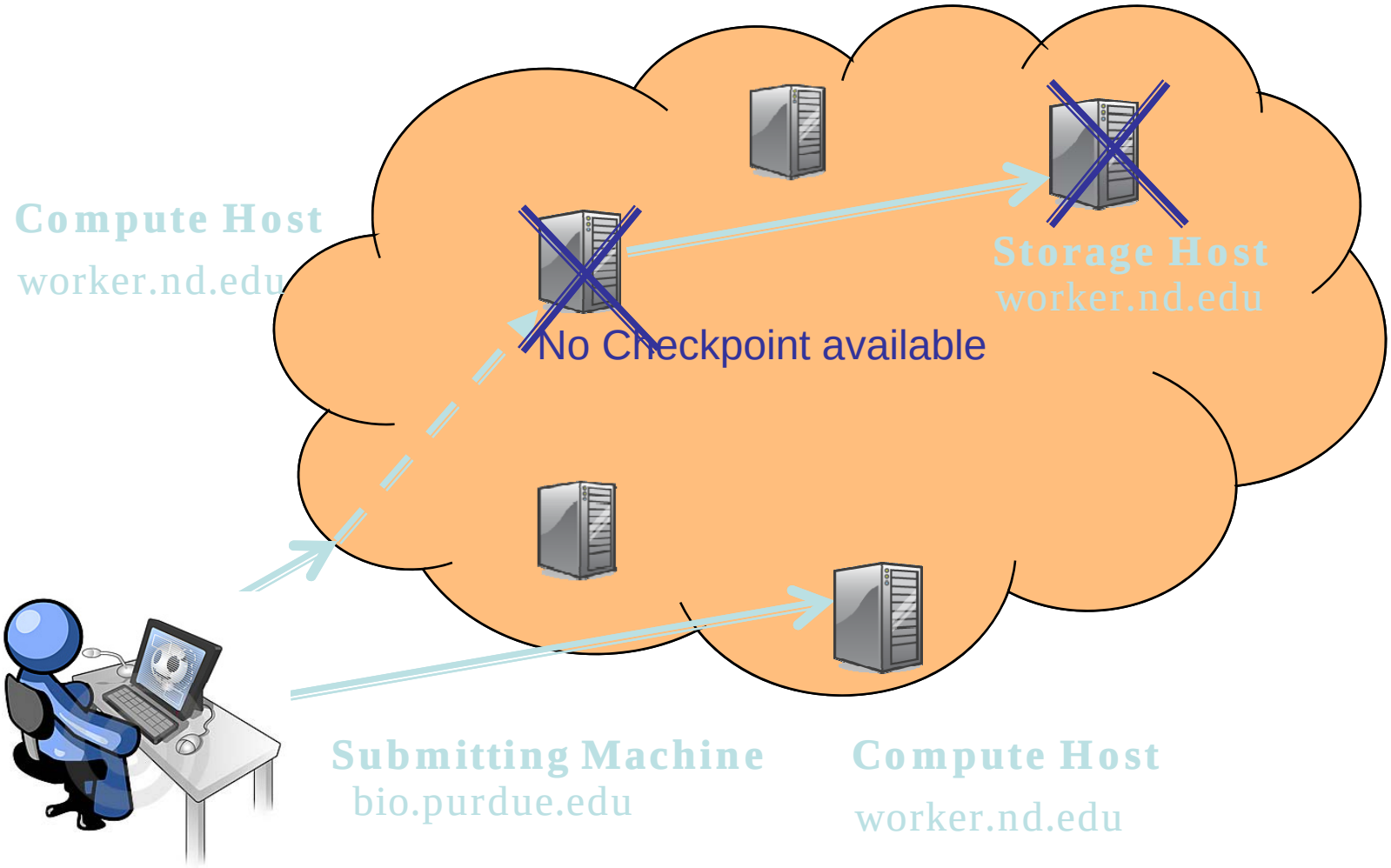
State of the Art Checkpoint-Recovery Scheme



Problem Motivation

- High overhead for application users
  - Submitting machine
    - If the submitter is behind a slow network (say, DSL modem)
  - Central storage server
    - High latency of transferring checkpoints back and forth between different university campuses (12% of the time)
    - High overhead when multiple machines are sending data to a single server
    - High overhead of sending data to a loaded server
- Stress on shared network resource
  - Transferring large amount of checkpoint data (gigabytes)
  - Transferring data across distant points in the network

Potential Solution and Challenges



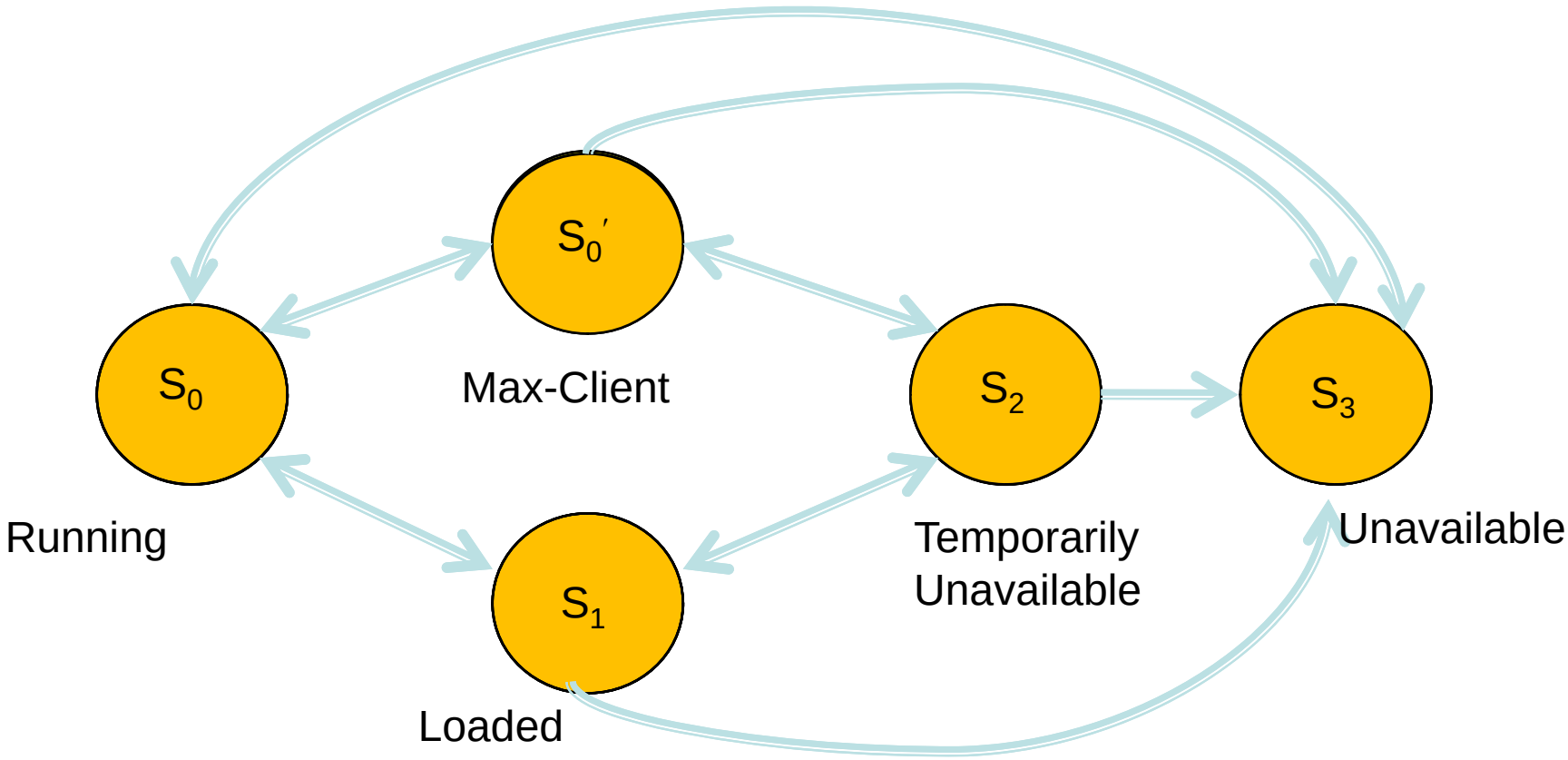
Shared Grid Environment

Contribution

- Goal: Can we improve the performance of the guest jobs by storing checkpoints in shared grid environment?
- Developed a reliable checkpoint-recovery system FALCON
  - Provides fault-tolerance through "Erasure Coding"
  - Selects reliable storage hosts which are nearby
  - Builds a failure model for storage hosts
  - Stores and retrieves checkpoints in efficient manner
- Deployed FALCON in BoilerGrid (DiaGrid)
- Performance improvement of benchmark applications in production grid is between 11% to 44%

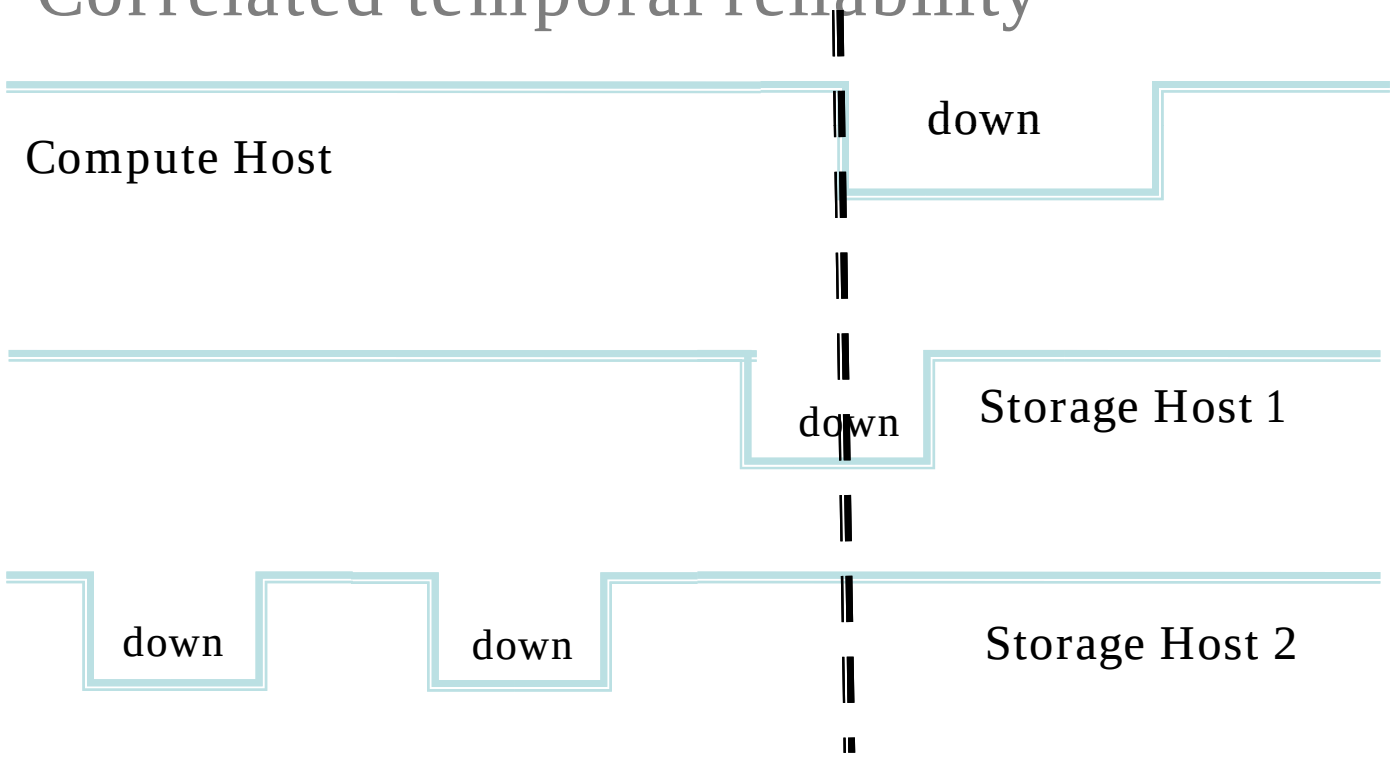
Failure Model

- Aids in predicting availability of the storage nodes
- Load: %utilization of I/O



Storage Repository Selection

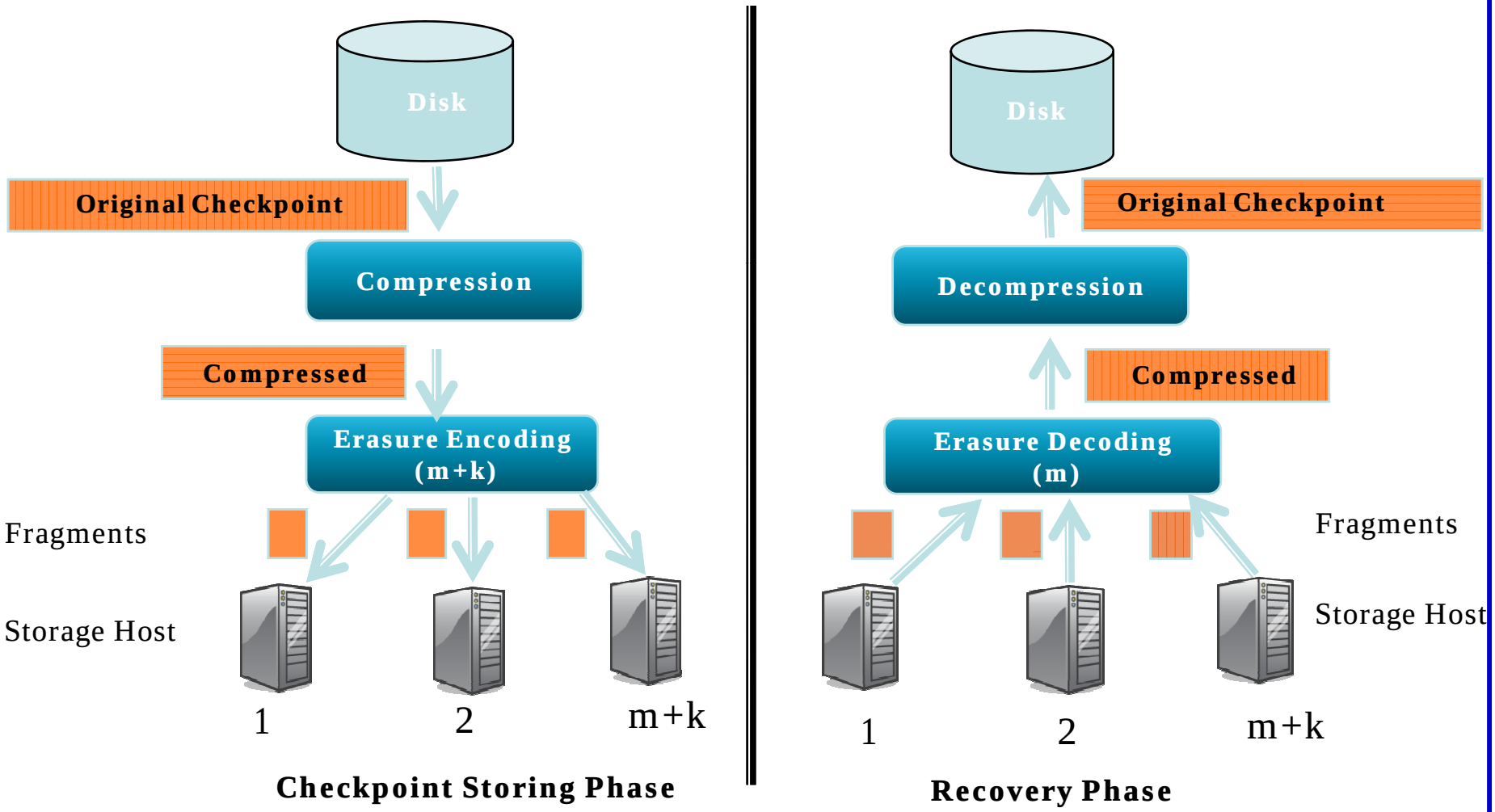
- Predict availability of storage nodes
  - Correlated temporal reliability



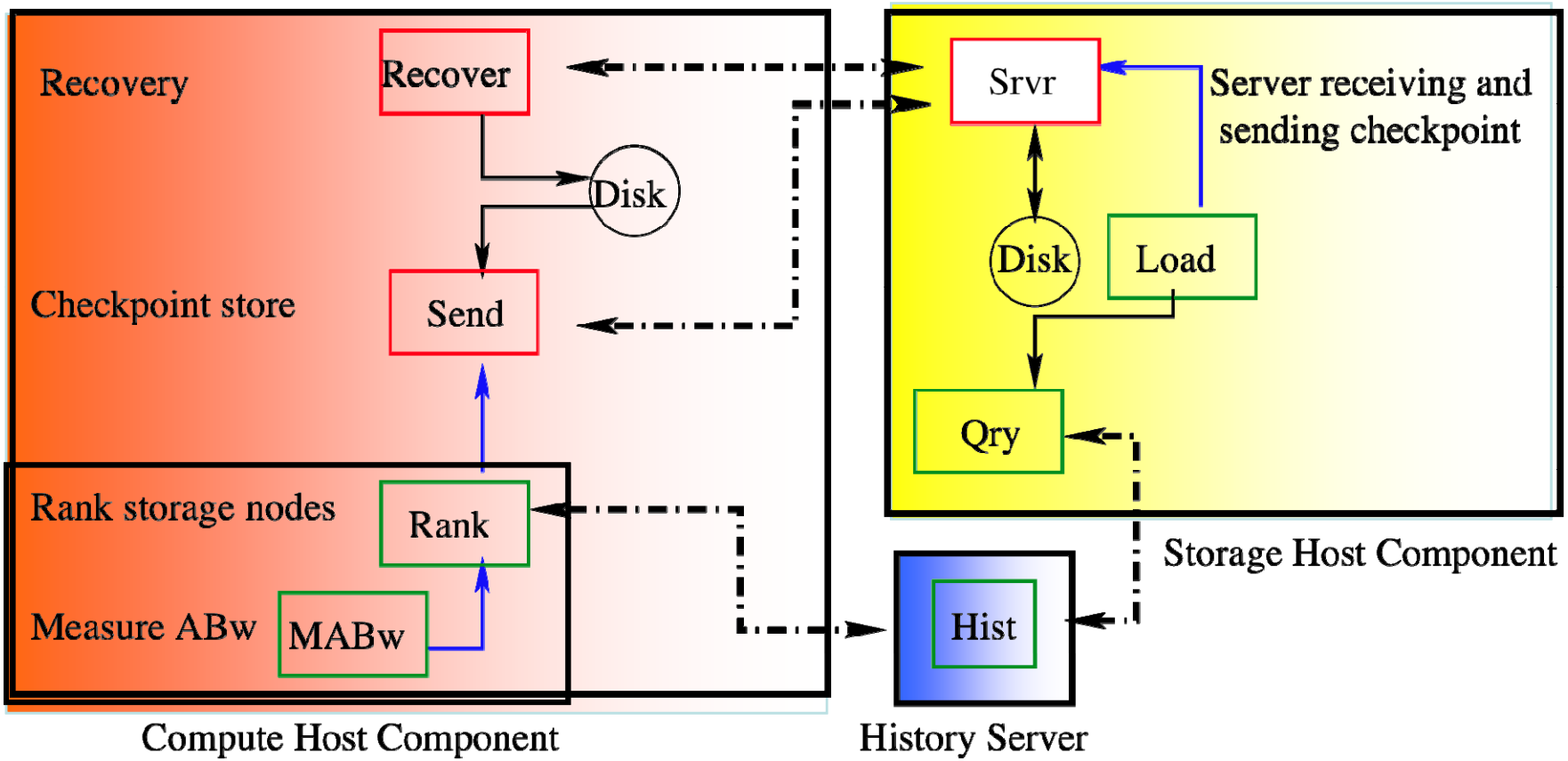
Storage Repository Selection

- Calculate network transfer overhead
  - Network Overhead = Amount of data to send (MB) / Available Bandwidth between a storage host and a compute host
- Minimize an objective function
  - Objective function: checkpoint storing overhead – benefit from the fact that a job can restart from the last saved state
  - Overhead includes network overhead
  - Benefit computed using the correlated temporal reliability
- Select a set of $m+k$ storage nodes that minimizes this objective function

Checkpoint-Recovery Scheme



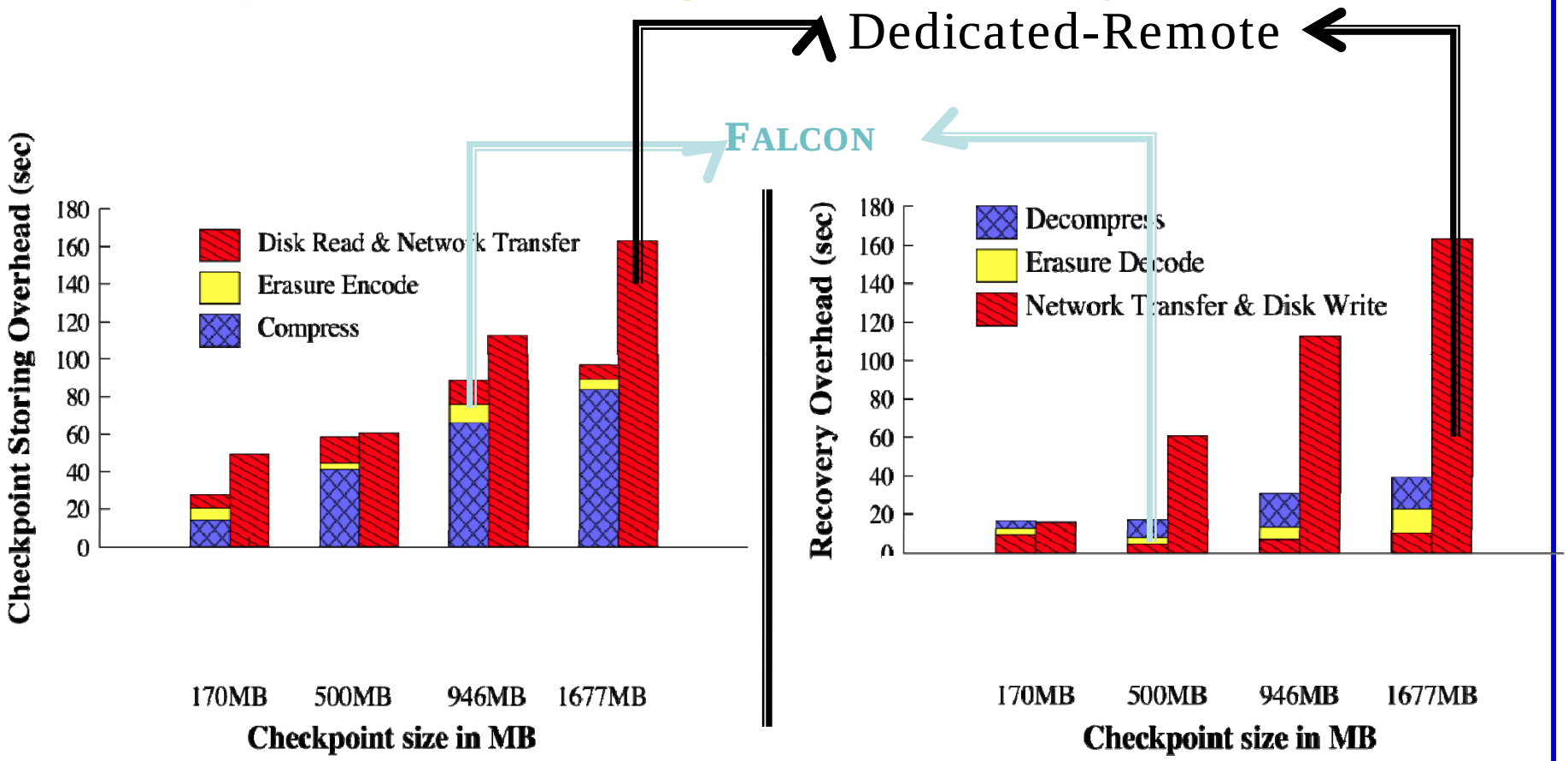
Structure of FALCON



Evaluation

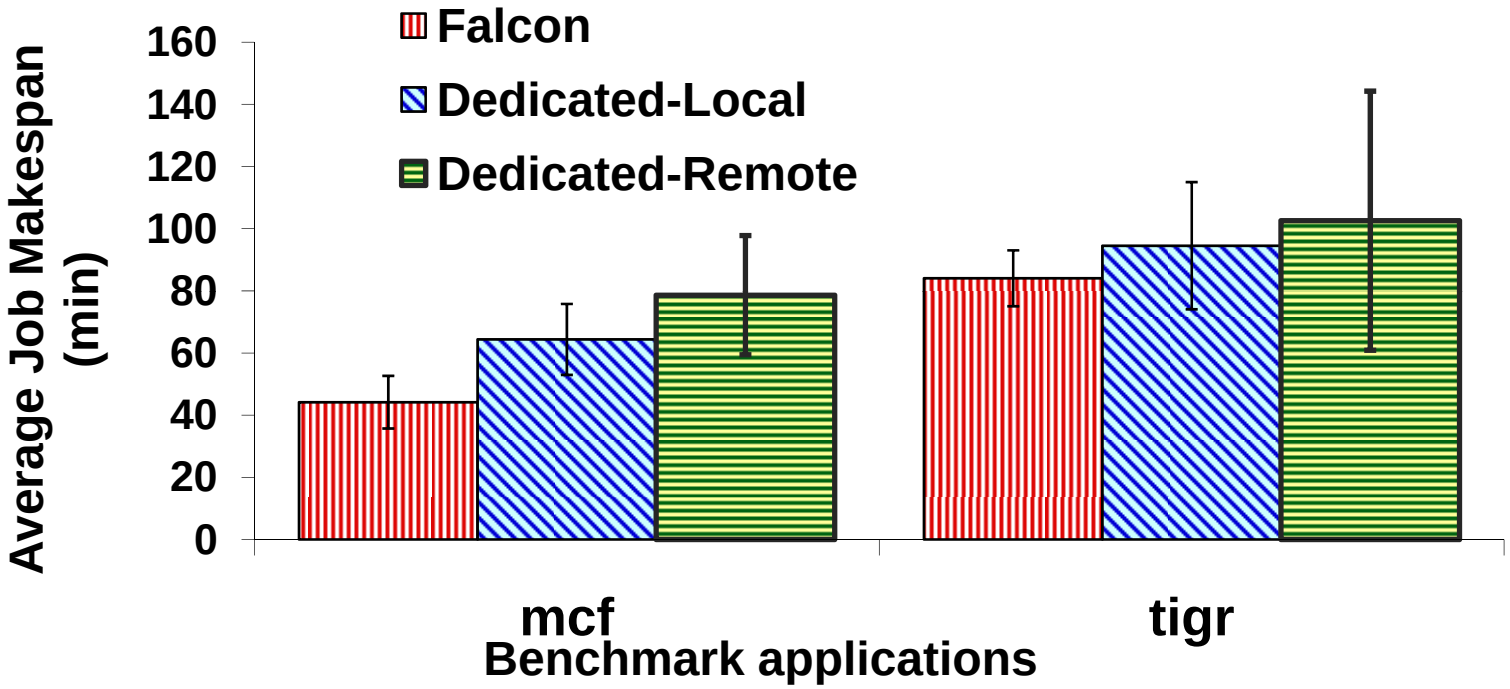
- Overall performance evaluation:
  - Average job makespan – the time a job takes to complete
- Efficiency of the checkpoint-recovery schemes:
  - Checkpoint storing overhead
  - Recovery Overheads
- Setup:
  - Submitted jobs to BoilerGrid
  - Applications – MCF (SPEC CPU 2006), TIGR (BioBench)
  - Erasure encoding parameters: m=3, k=2

Checkpoint Storing & Recovery Overhead



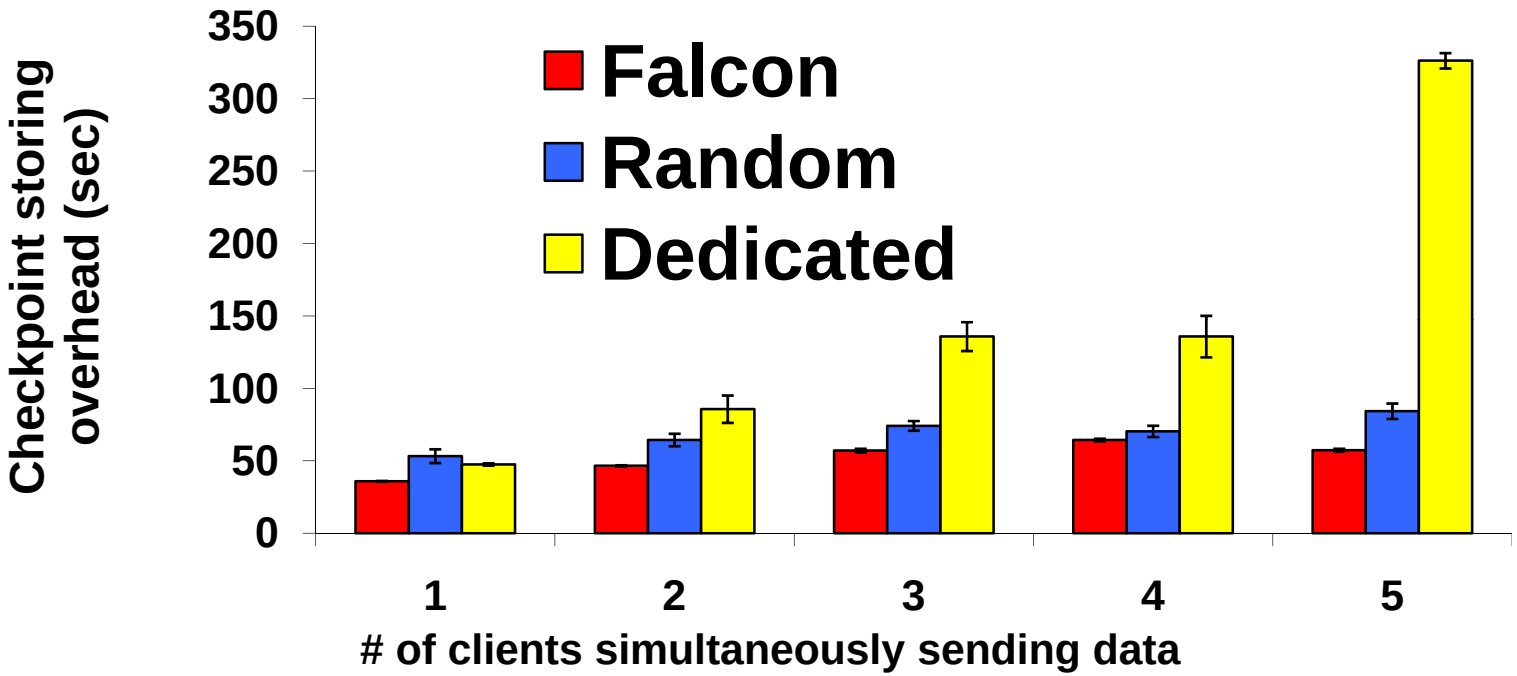
- Performance of Falcon scales with the increase in the checkpoint sizes
- Lower network transfer overhead and lower utilization of shared network bandwidth

Overall Performance Comparison



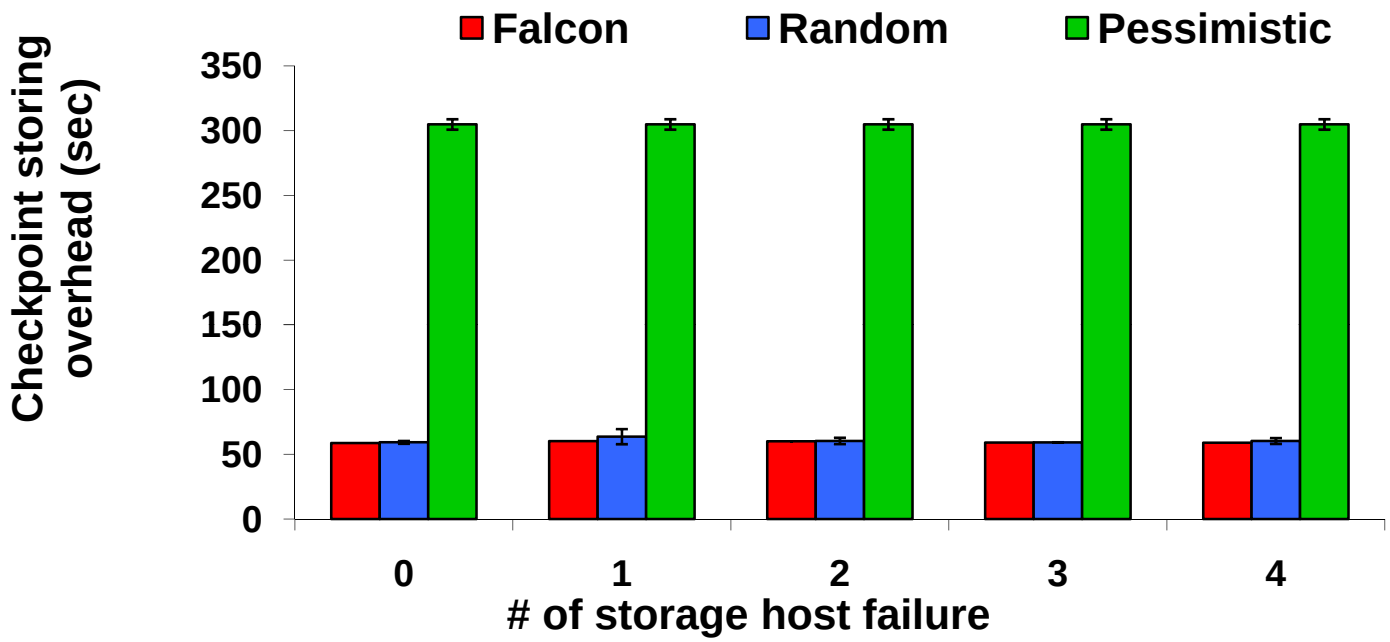
- Performance improvement of the applications are between 11% and 44%

Handling Simultaneous Clients



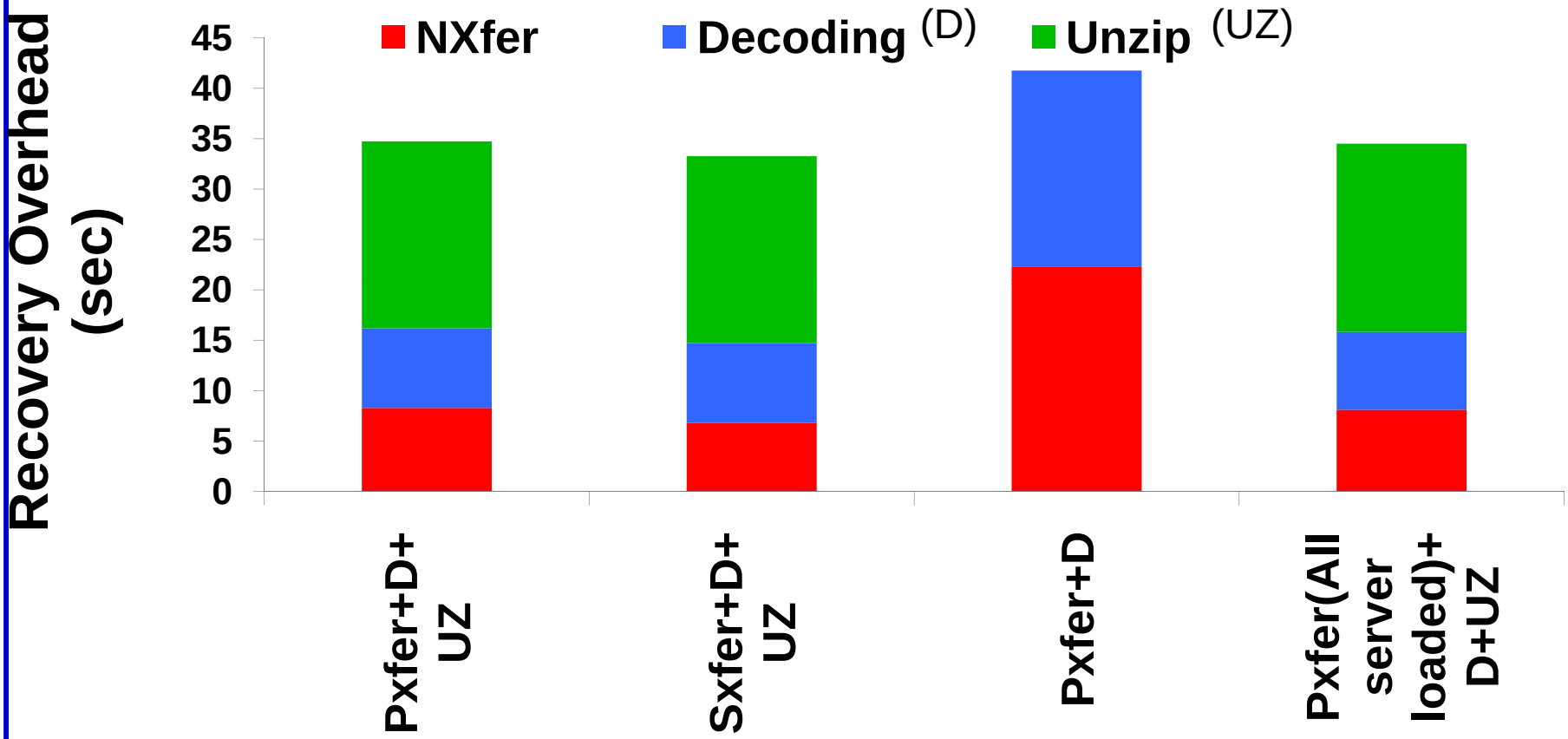
- Performance of dedicated scheme suffers
- Performance of Random scheme suffers because of choosing machine behind slow network

Handling Storage Failures



- Robustness at no extra cost for Falcon
- Pessimistic incurs large overhead

Contributions of Components



- Pxfer – parallel network transfer, Sxfer – serial network transfer
- Largest contribution comes from compression

Conclusion

- For practical FGCS systems, runtime prediction of resource unavailability is important
- Resource unavailability may occur due to resource contention or resource revocation
- Prediction system based on an SMP is
  - Fast: < 0.006% overhead
  - Accurate: > 86% accuracy in average
  - Robust: < 6% difference caused by noise
- Prediction system helps a runtime scheduler
- For handling job evictions, we need checkpoint-restart
  - It is possible to use shared machines as checkpoint repositories
  - Dynamic selection of checkpoint repositories reduces application makespan

Ongoing Work

- If we have parallel file system to store checkpoints, how can we reduce its load
  - Exploit the similarities in checkpoints of similar processes
  - Aggregate and compress multiple similar checkpoints
  - Perform this in a tree-based form that scales with the number of processes
- What kinds of applications lend themselves to the shared and optimized checkpointing

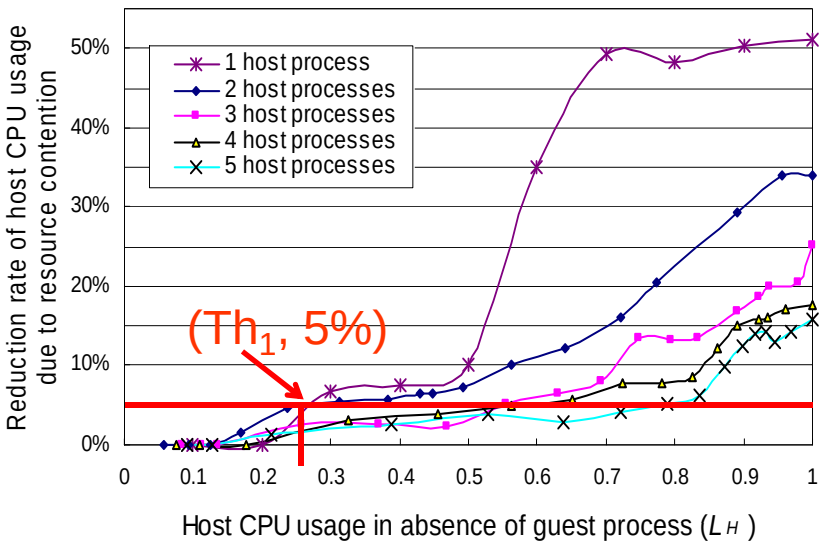
Thanks!

Backup Slides

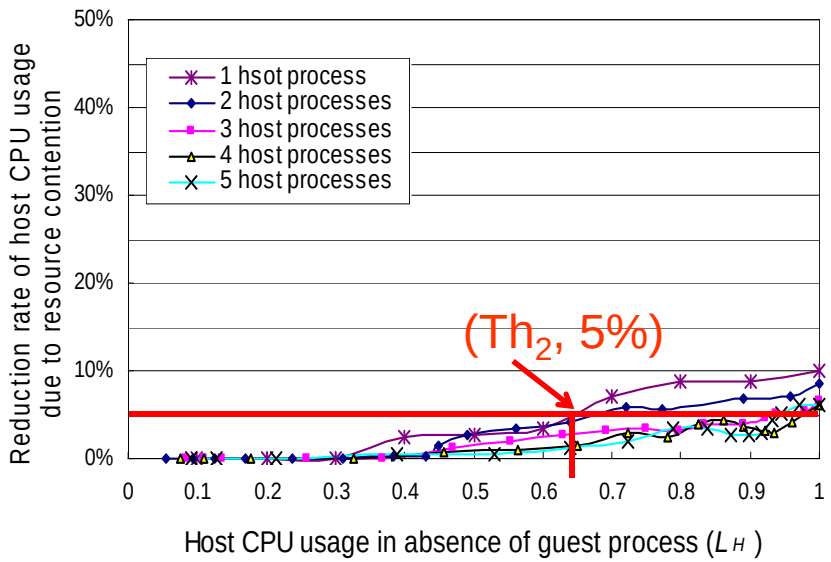
- Resource contention studies, 27-29
- Linux scheduler, 30
- Details on reference algorithms for failure prediction, 31

Empirical Studies on Resource Contention

- CPU Contention
  - CPU-intensive guest applications
  - host groups consisting of multiple processes with diverse CPU usage
  - 1.7 GHz Redhat Linux machine

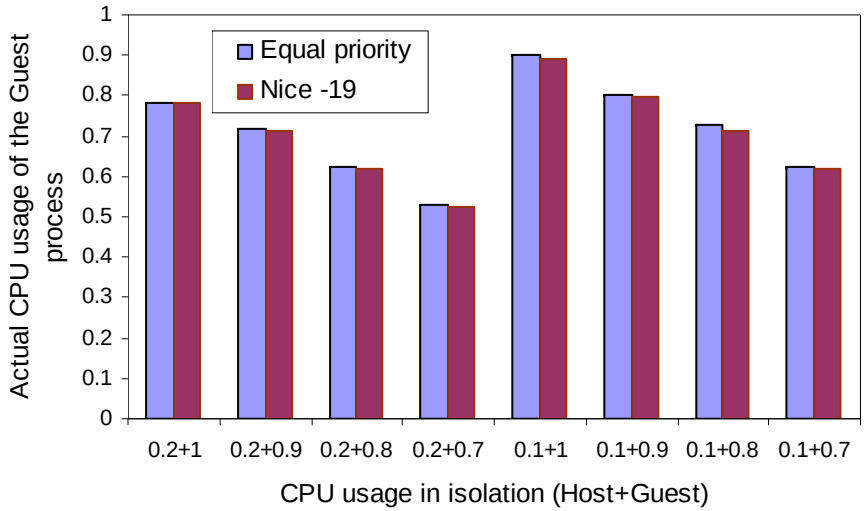


All processes have the same priority

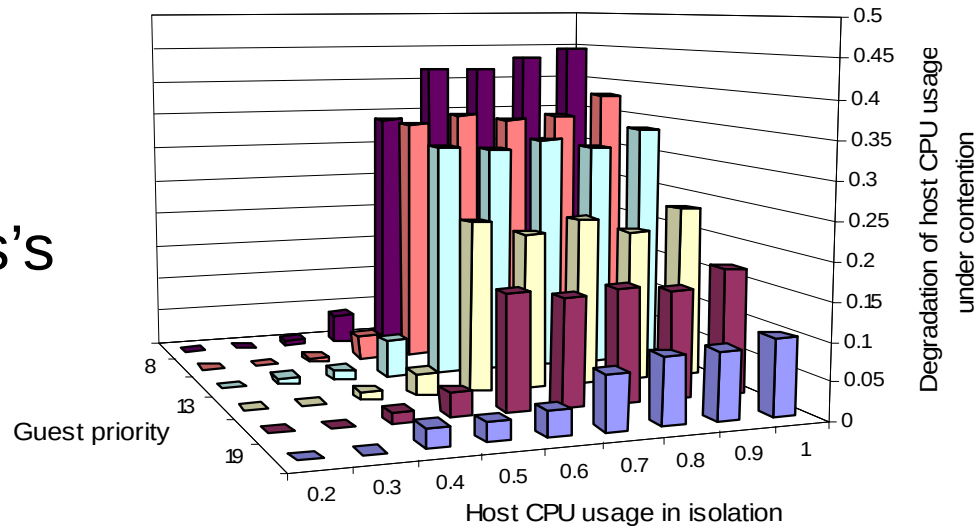


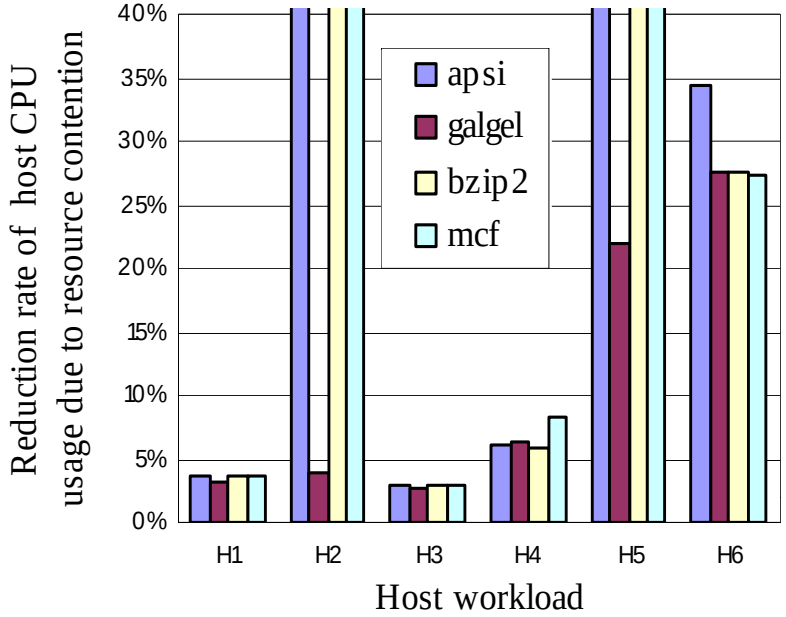
Guest process takes the lowest priority

Restrict resource contention by minimizing guest process's priority from its creation

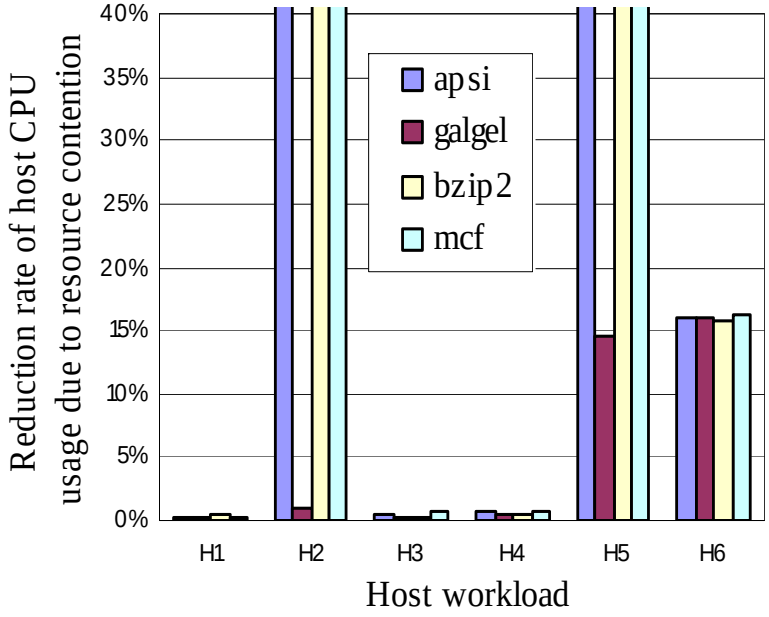


Restrict resource contention by finely tuning guest process's priority





Guest process with priority 0



Guest process with priority 19

- Memory thrashing happens when processes desire more memory than the system has
- Impacts of CPU and memory contention can be isolated
- The two thresholds, Th_1 and Th_2, can still be applied to quantify CPU contention

```
While (1) {
   If exists p such that p.state = RUNNABLE
      Foreach process p
         p.quanta = 20 + p.niceLevel + 1/2 * p.quanta;
   While exists a process p
      such that (p.state = RUNNABLE) and (p.quanta > 0)
         Select p with largest p.quanta;
         Decrement p.quanta;
         Run p;
}
```

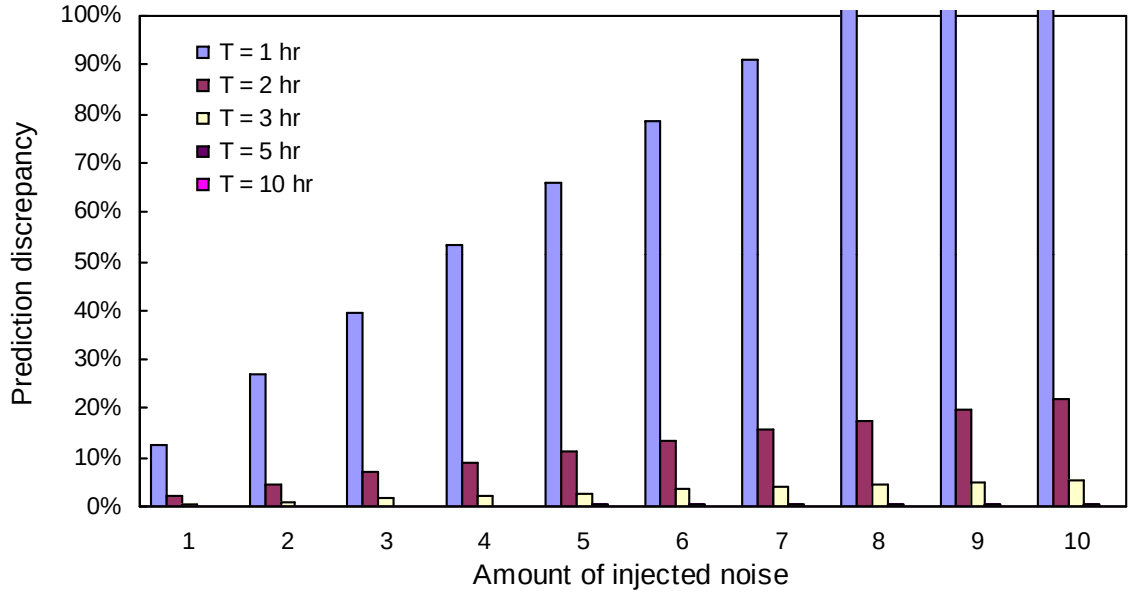
Linux CPU scheduler

Details on Reference Algorithms

- AR(p) – An autoregressive model is simply a linear regression of the current value of the series against one or more prior values of the series. p is the order of the AR model. Linear least squares techniques (Yule-Walker) are used for model fitting.
- BM(p) – Average on previous N values. N is chosen to minimize the squared error
- MA(p) - A moving average model is conceptually a linear regression of the current value of the series against the white noise or random shocks of one or more prior values of the series. Iterative non-linear fitting procedures (Powell's methods) need to be used in place of linear least squares.
- ARMA(p,q) - a model based on both previous outputs and their white noise
- LAST – the previous observations from the last time window of the same length are used for prediction

Prediction Robustness

Randomly insert unavailability occurrences between 8:00-9:00 am on a weekday trace



1) Predictions on smaller time windows are more sensitive
2) On large time windows (> 2 hours), intensive noise (10 occ urrences within one hour) causes less than 6% disturbance in the prediction

Summary on Related Work

- Fine-grained cycle sharing with OS kernel modification Ryu and Hollingsworth, *TPDS*, 2004
- Critical event prediction in large-scale clusters Sahoo, et. al., *ACM SIGKDD*, 2003
- CPU load prediction for distributed compute resources Wolski, et. al., *Cluster Computing*, 2000
- Studies on CPU availability in desktop Grid systems Kondo, et. al., *IPDPS*, 2004