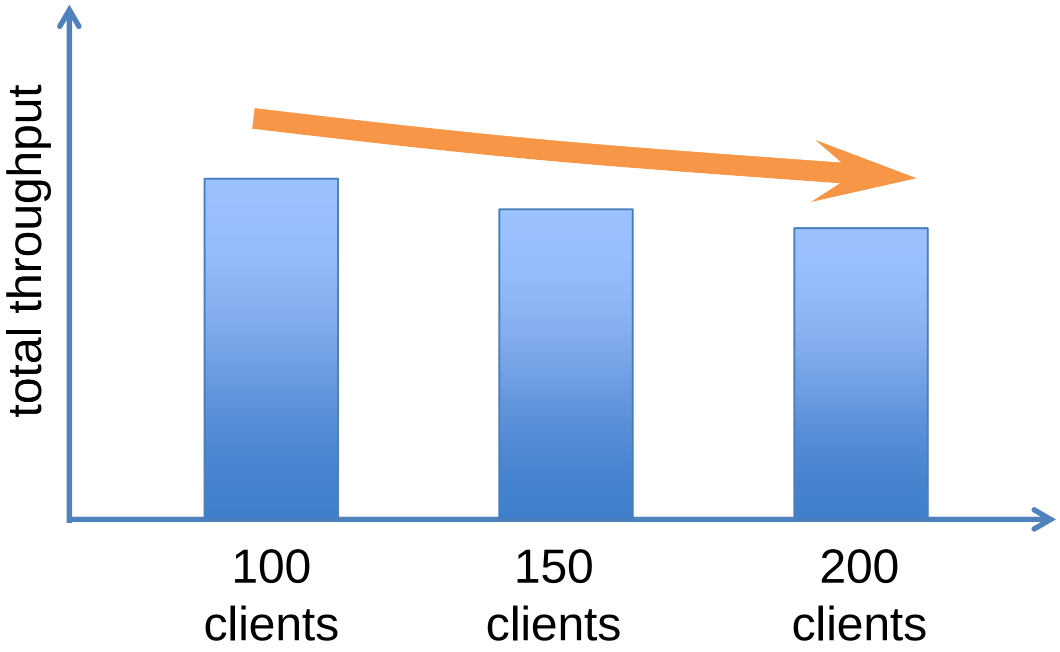
Increasing Performance Through Automated Contention Management

(LUG'16)

Yan Li, Xiaoyuan Lu,
Ethan Miller, Darrell Long
Storage Systems Research Center (SSRC)
University of California, Santa Cruz
(a Intel® Parallel Computing Center)

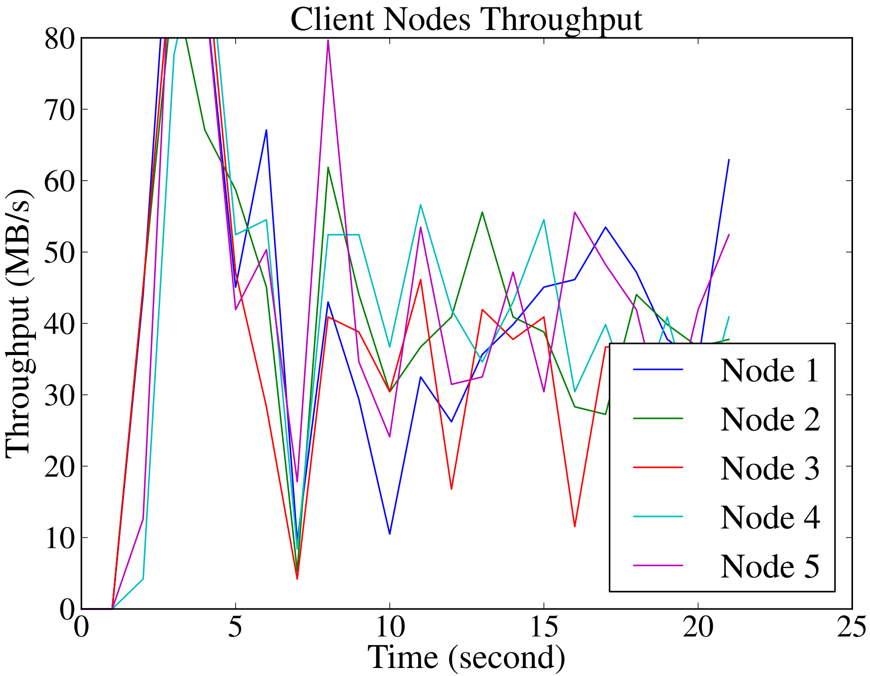
Challenge: consistent performance at peak times

congestion harms *efficiency* and *throughput*



Challenge: consistent performance at peak times

congestion causes *fluctuation*



client throughput of a random write workload

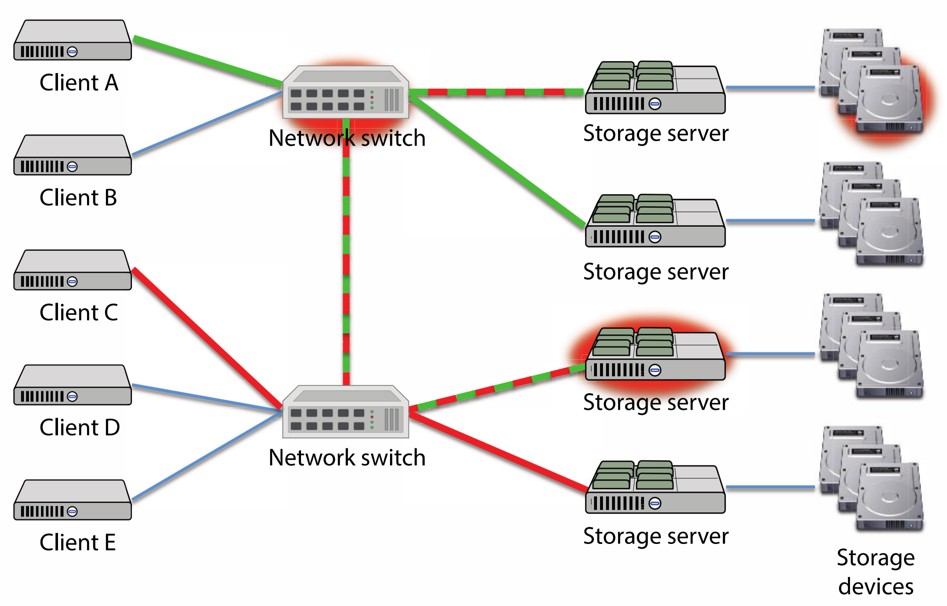
5 nodes accessing 5 servers

ascar.io

ssrc

Challenge: consistent performance at peak times

congestion can occur *anywhere* in the cluster



ascar.io
ssrc

The problem we are trying to solve

Picture credit:
https://www.checkfelix.com/reiseblog/10-argumente-urlaub-schottland/

The problem we are trying to solve

Improve throughput or fairness during congestion

or both at the same time!

End-to-end coverage

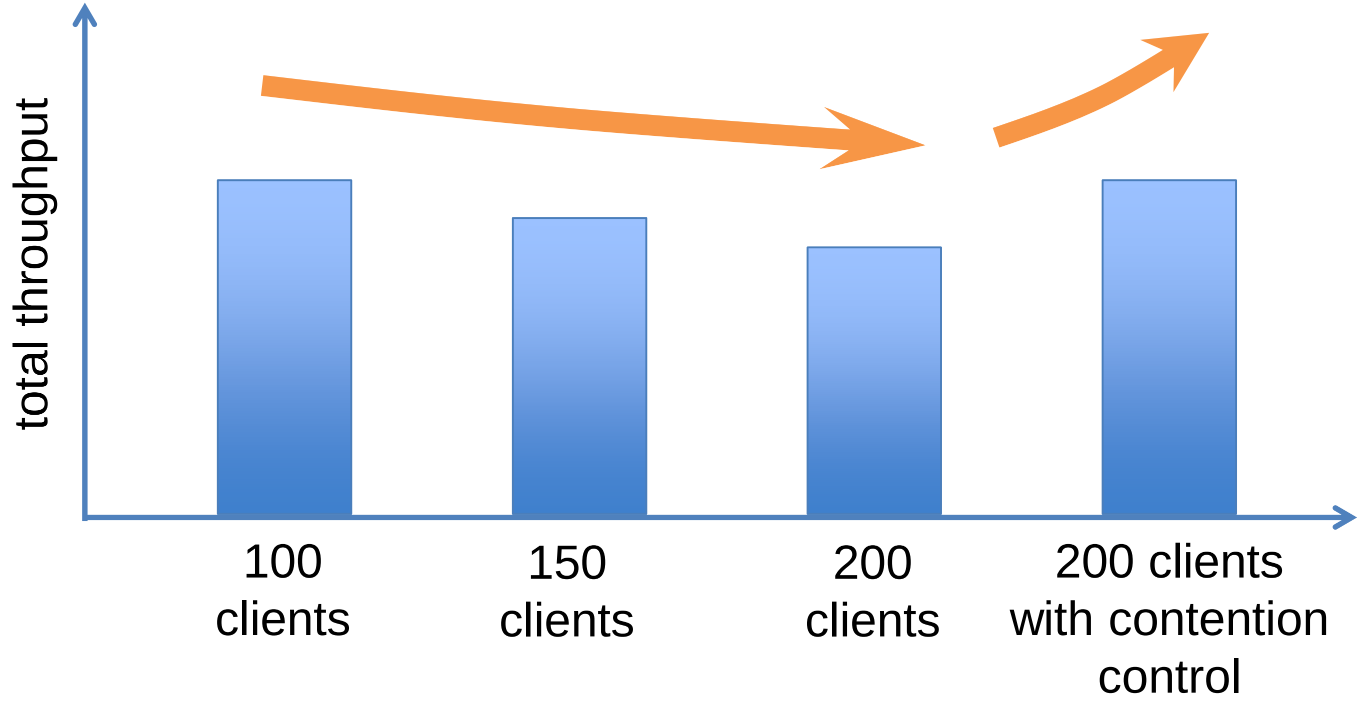
handling congestion at OSC, network, OSS, and OST

Fully automatic and requires little human effort

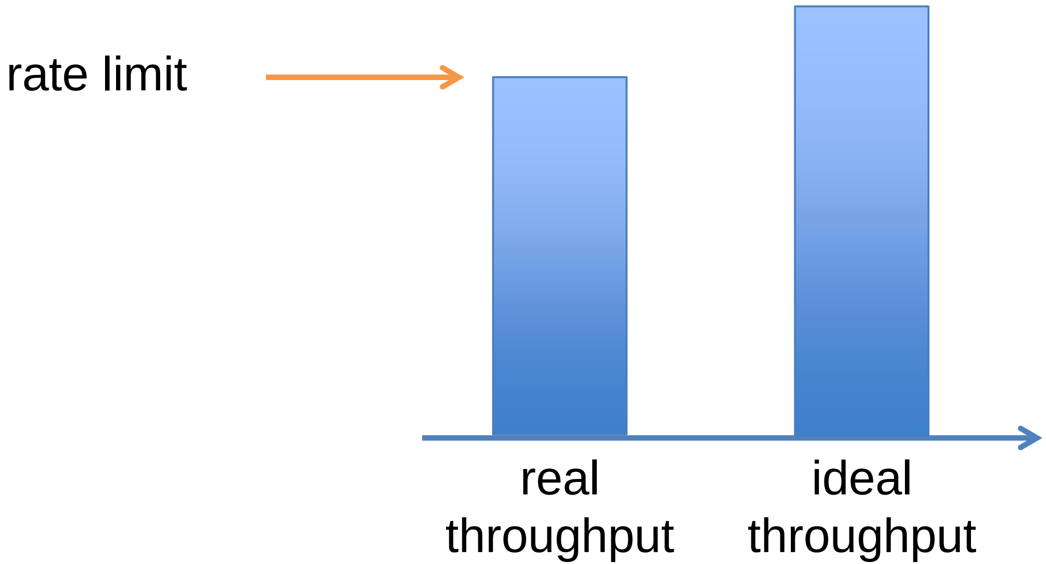
modern systems are very dynamic, and we won’t have time to create models

Rate limiting can improve performance

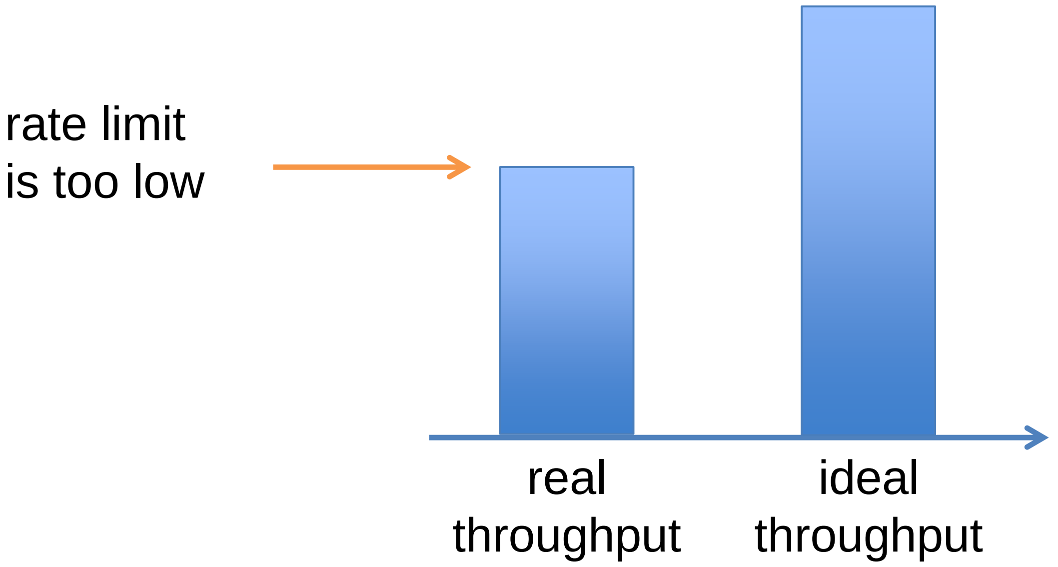
... if done properly



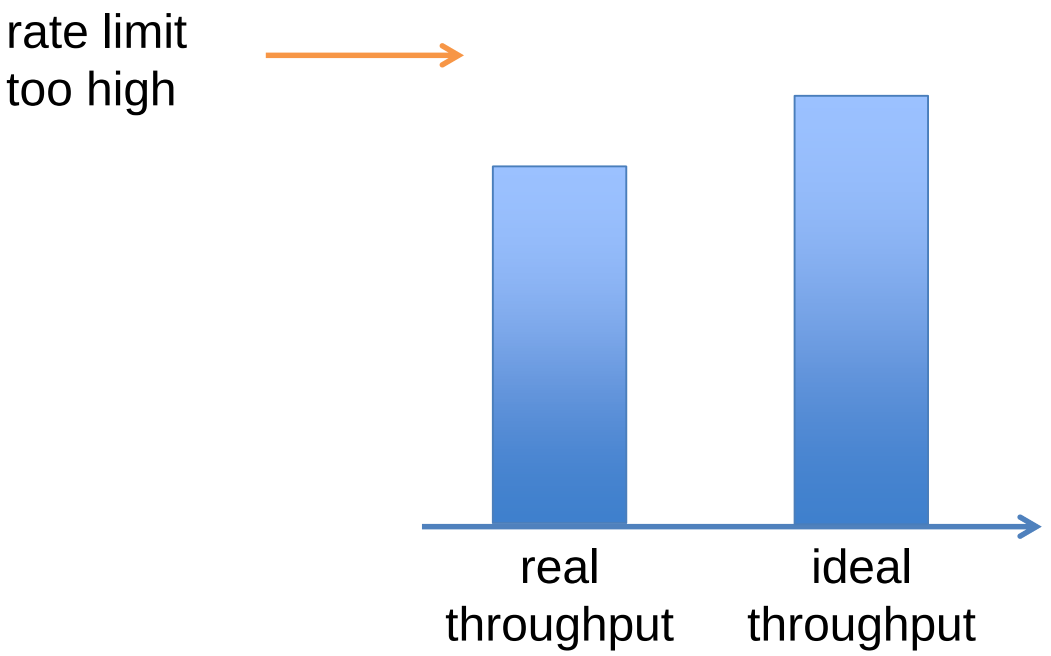
Challenges of distributed I/O rate control:
1. Where is the *sweet spot*?



Challenges of distributed I/O rate control: 1. Where is the *sweet spot*?



Challenges of distributed I/O rate control: 1. Where is the *sweet spot*?

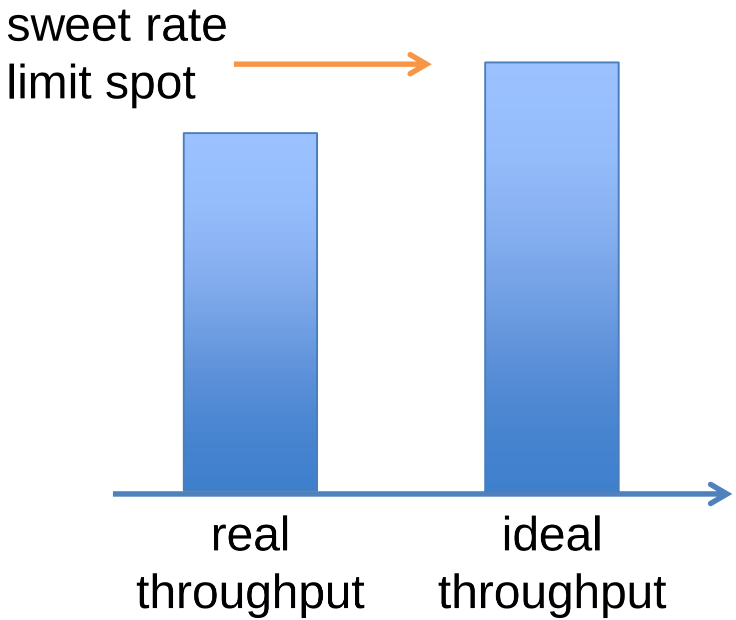


ascar.io ssrc

Challenges of distributed I/O rate control: 1. Where is the *sweet spot*?

Capability discovery usually involves communication:

- between clients
- with a central controller



Challenges of distributed I/O rate control: 2. scalability

Intra-node communication can grow at $O(n^2)$

Adds overhead to already congested network

Low responsiveness for highly dynamic workload

ASCAR: Automatic Storage Contention Alleviation and Reduction

Client-side rule-based I/O rate control

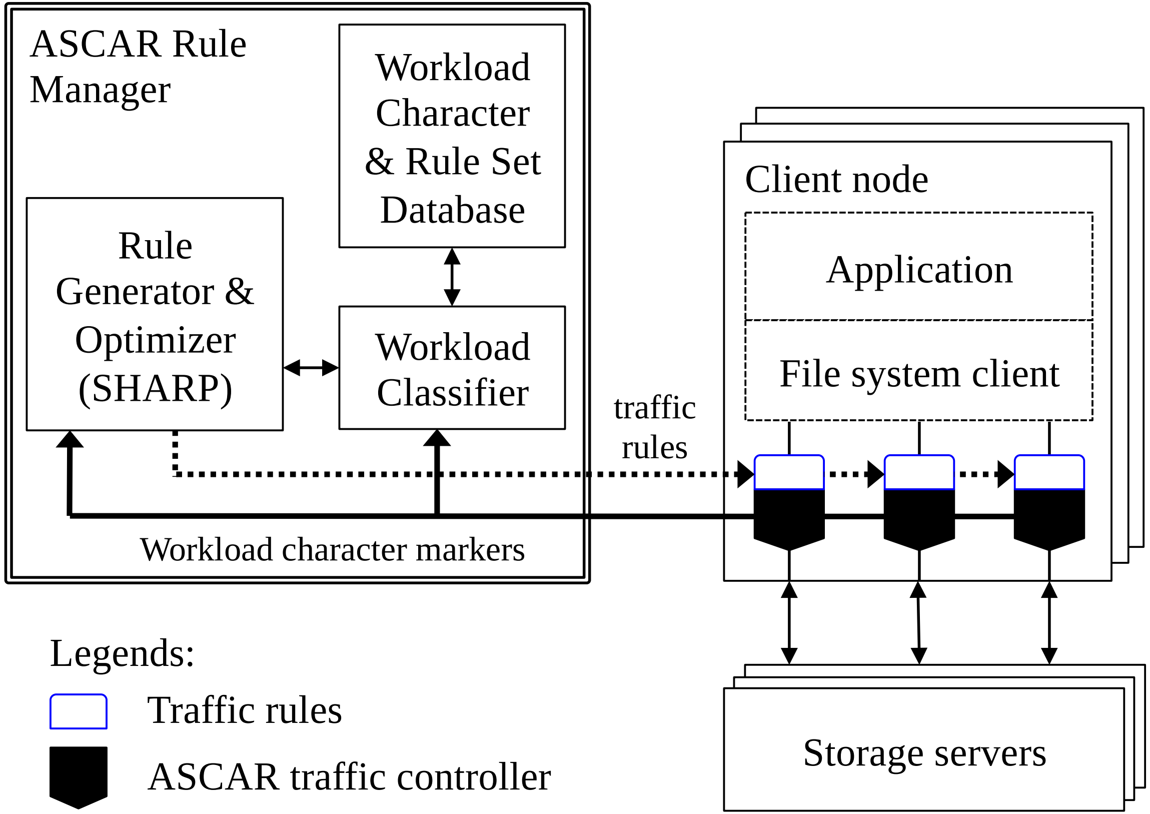
1. no need for central scheduling or coordination, nimble and highly responsive
2. no need to change server software or hardware
3. no scale-up bottleneck

Use machine learning and heuristics for rule generation and optimization

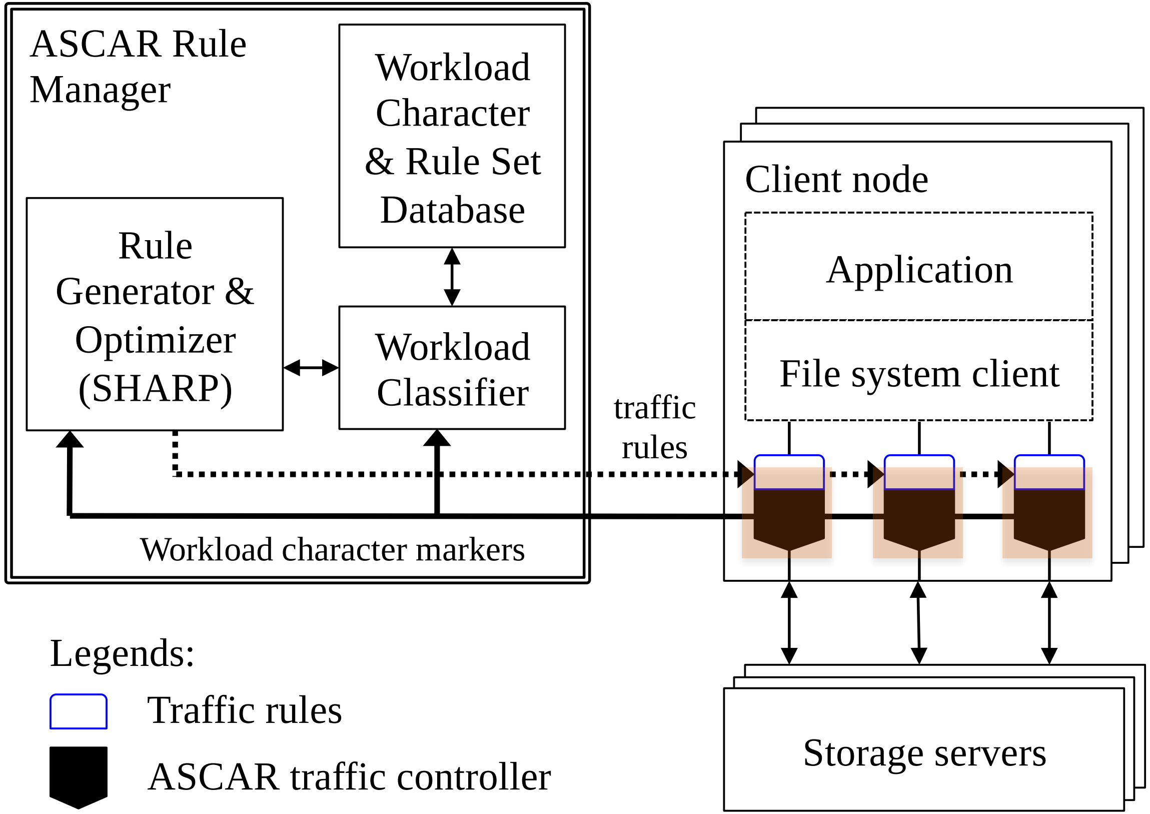
no prior knowledge of the system or workload is required

ascar.io

Components of the ASCAR prototype

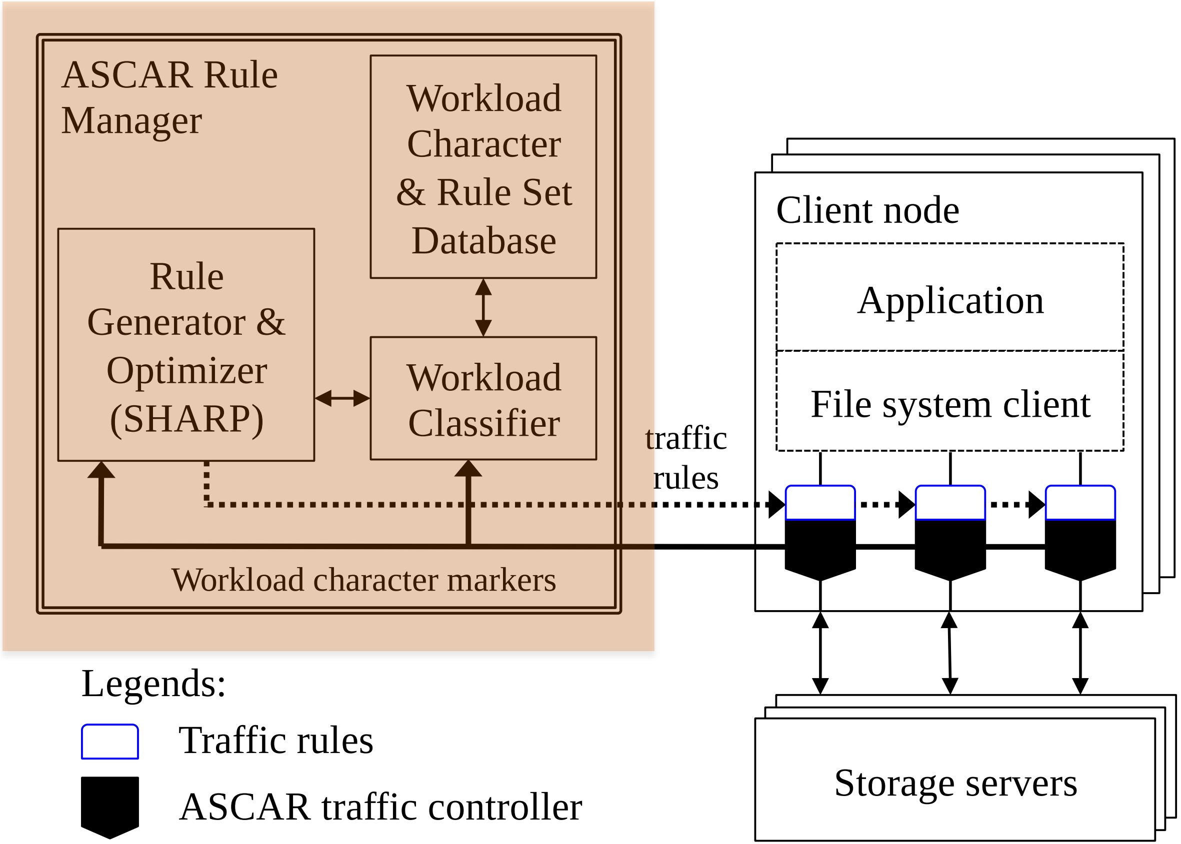


Components of the ASCAR prototype



ascar.io
ssrc

Components of the ASCAR prototype



Rule-based Contention Control

Rules tell the controller how to react to congestions
tweak the congestion window according to request processing latency

Each client tracks three congestion state statistics
(ack_ewma, send_ewma, pt_ratio)

Each rule maps a congestion state to an action
(Congestion State (CS) statistics) → <action>

An action describes how to change the congestion window (max_rpcs_in_flight) and rate limit:<m, b, τ>
new_cong_window = m × cong_window + b
τ is the rate limit

Generating a rule set for a certain workload

Extract a short signature workload that covers the important features of the application's I/O workload

a signature workload is usually 20 ~ 30 seconds long

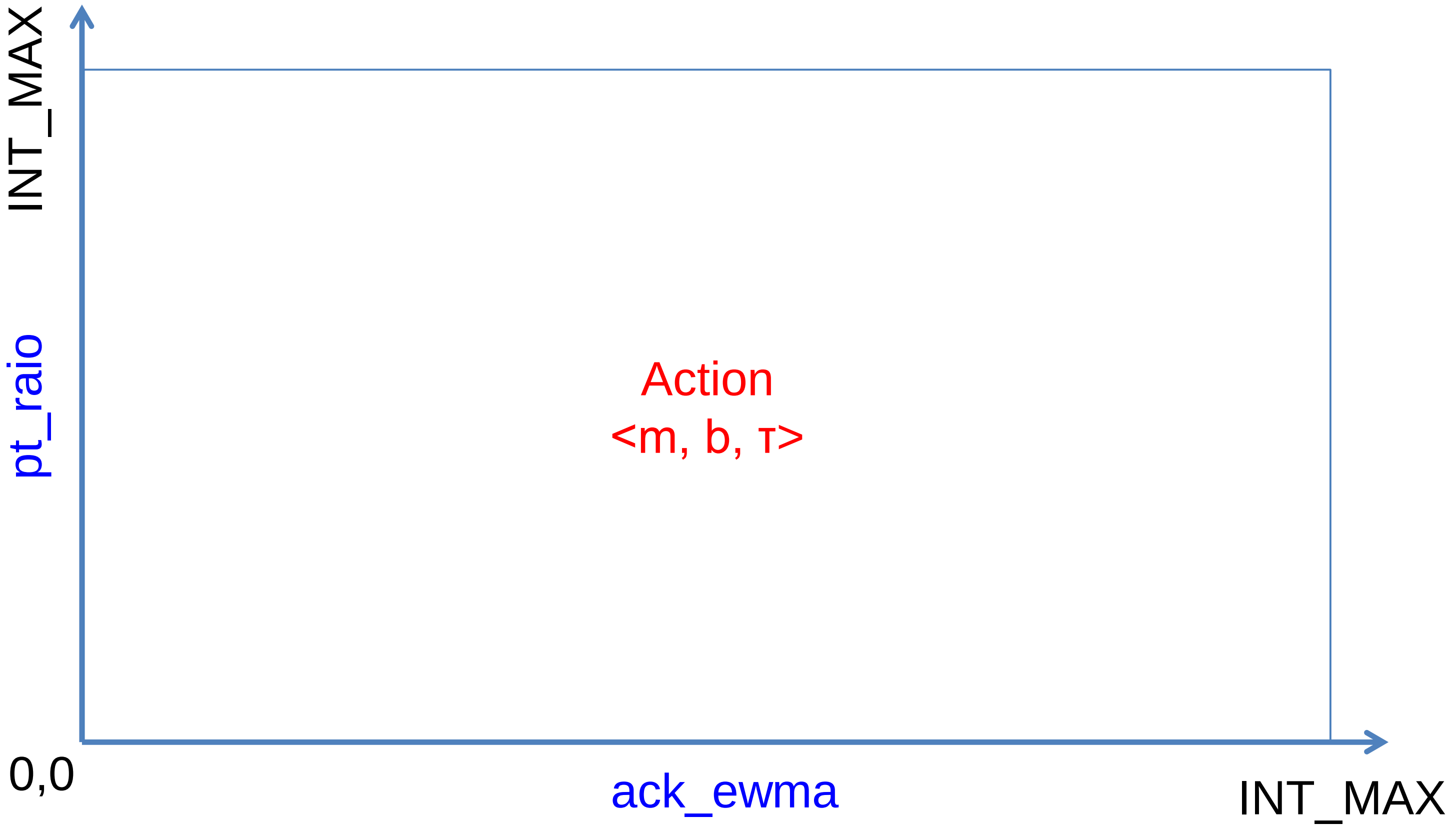
Generate candidate rules and benchmark them with the signature workload on the real system

test possible combinations of action variables

Split the hottest rule in the set to generate more rules

structural improve vs. tuning parameters

Begin with one rule: the whole state space maps to one action



Try different values of <m, b, τ> with the workload

INT_MAX

pt_raio

Try the Cartesian product of:

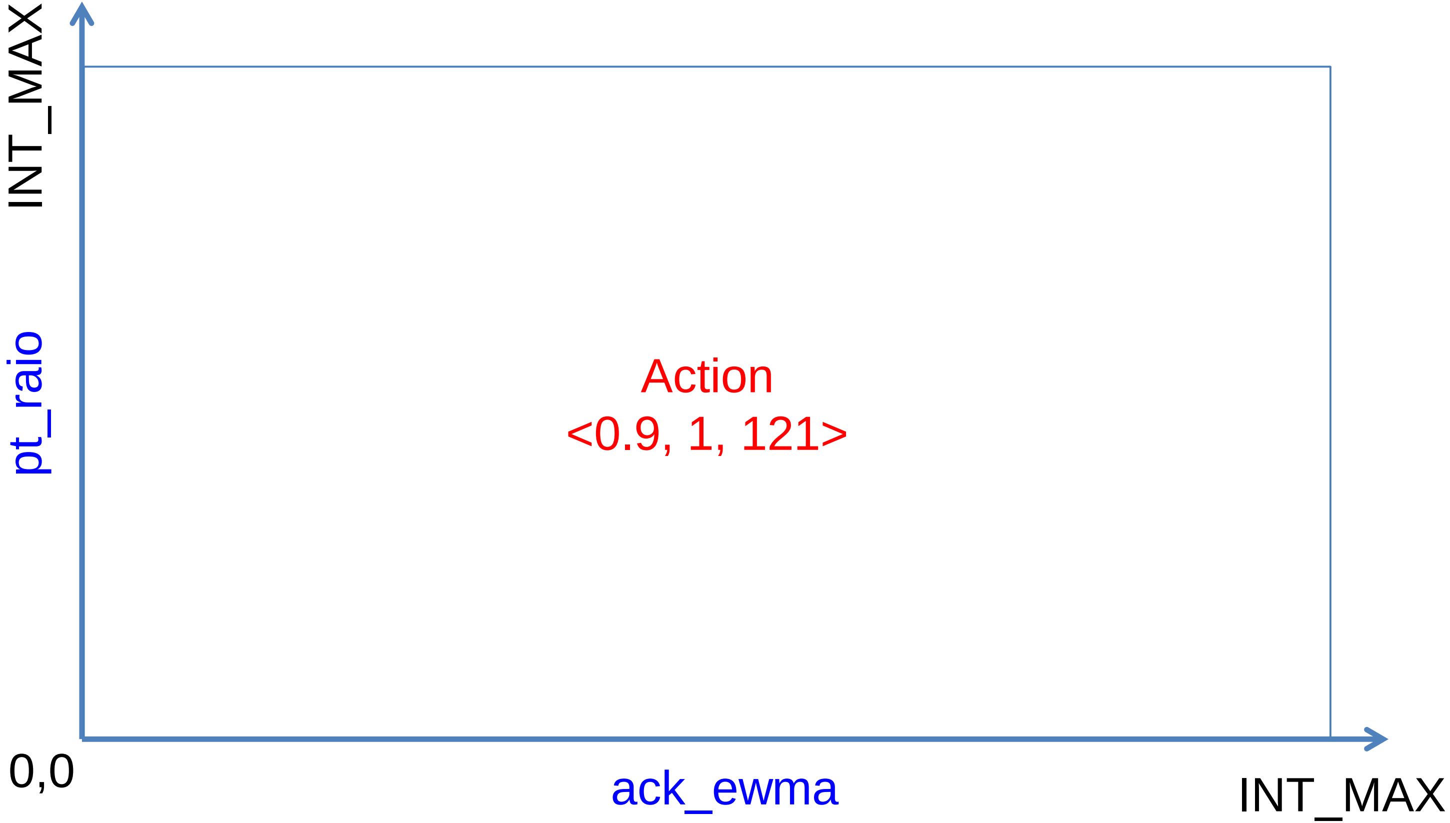
{m ± 0.01, m ± 0.02, m ± 0.04, … }
×
{b ± 0.01, b ± 0.02, b ± 0.04, … }
×
{ τ ± 1, τ ± 2, τ ± 4, … }

0,0

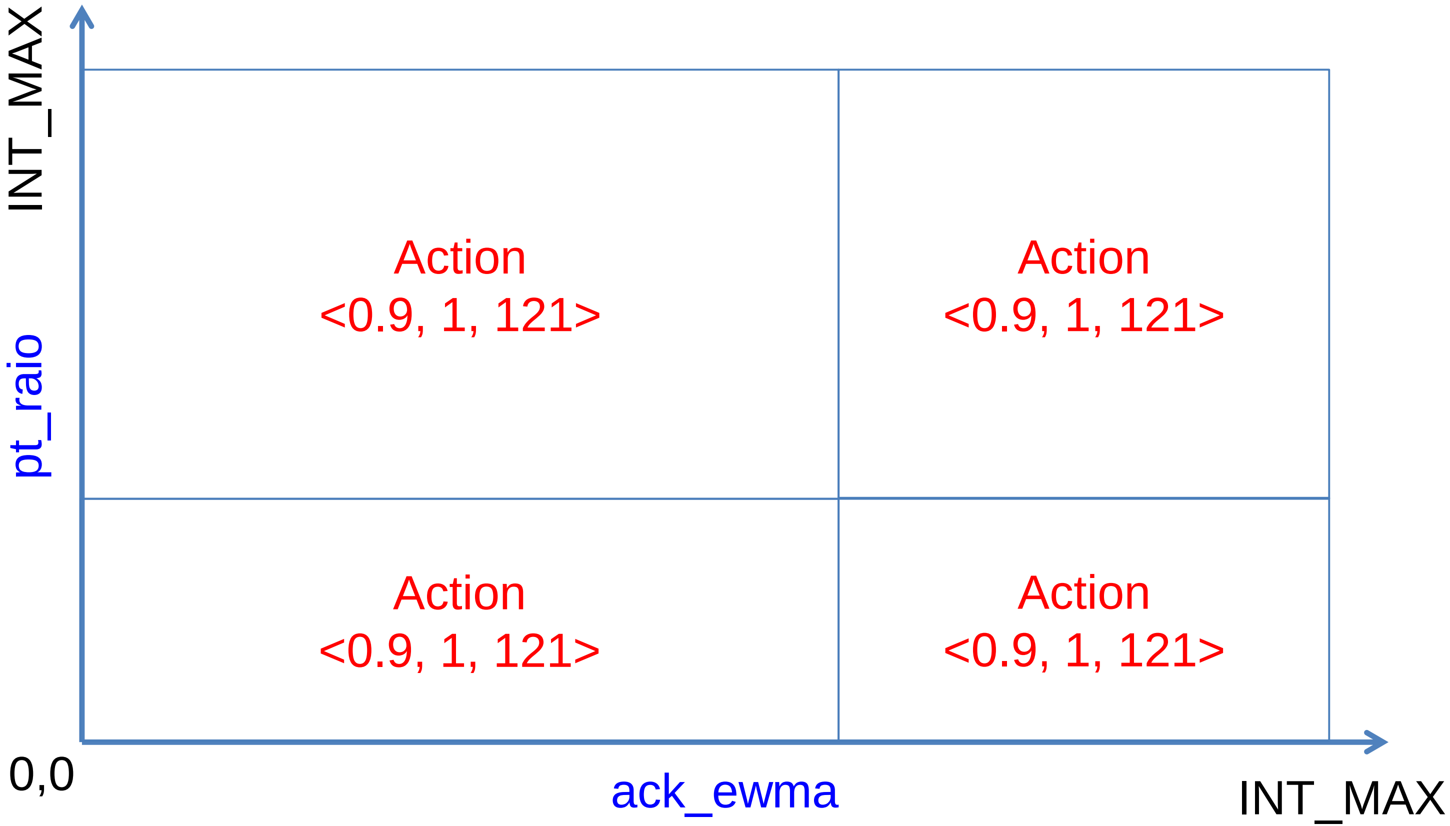
ack_ewma

INT_MAX

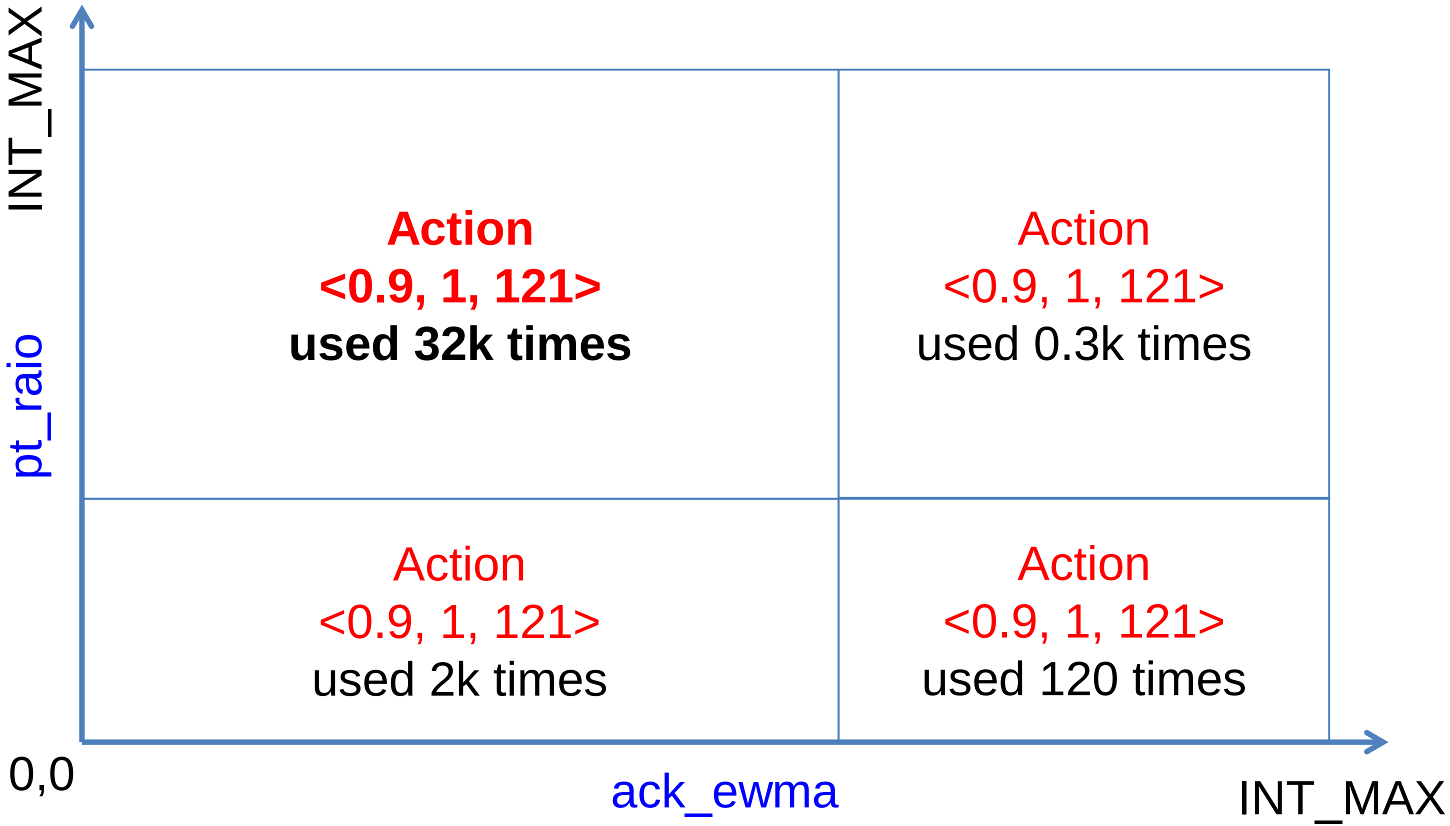
Find the rule that yields highest performance



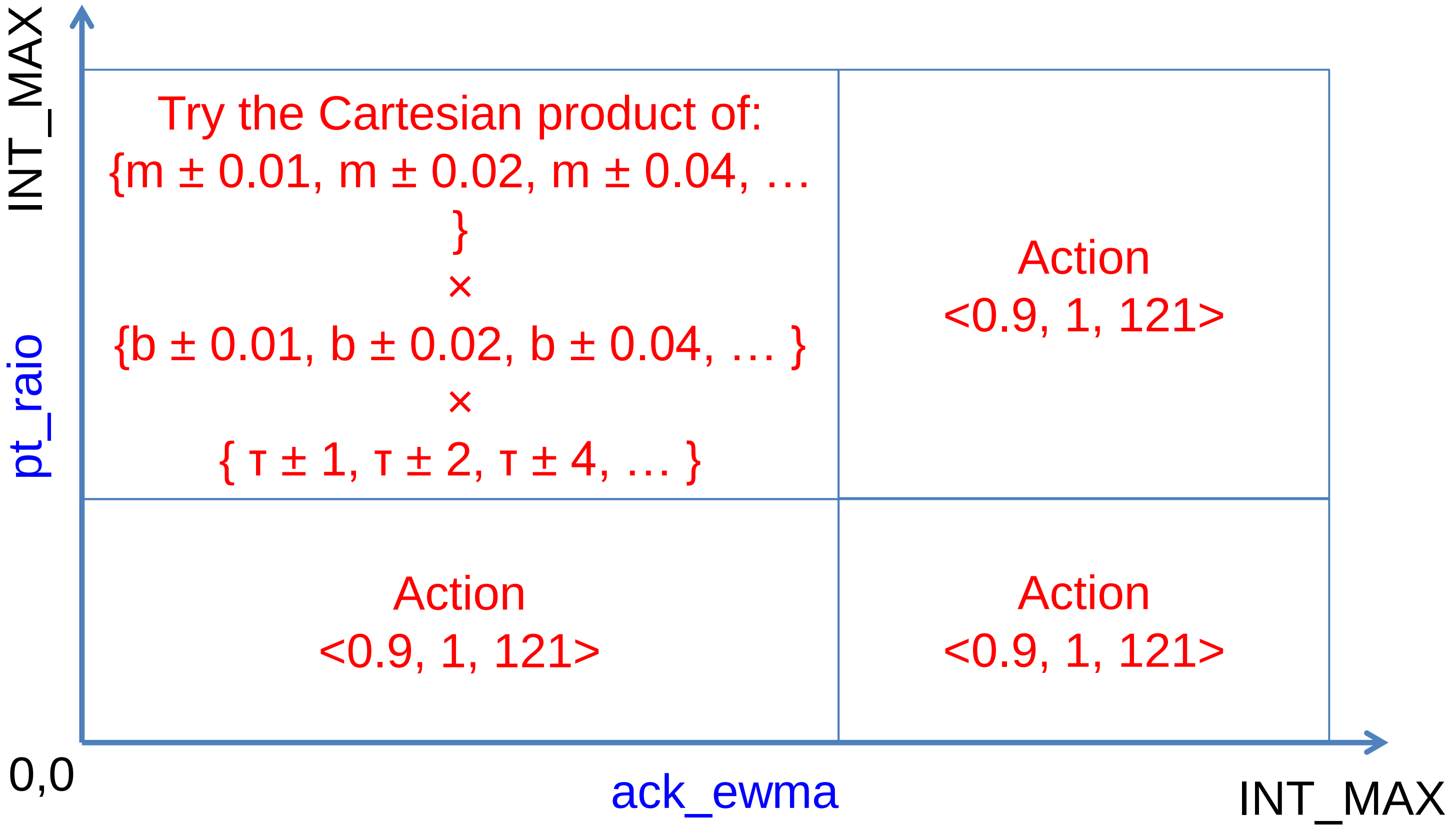
Split the state space at the most observed state values



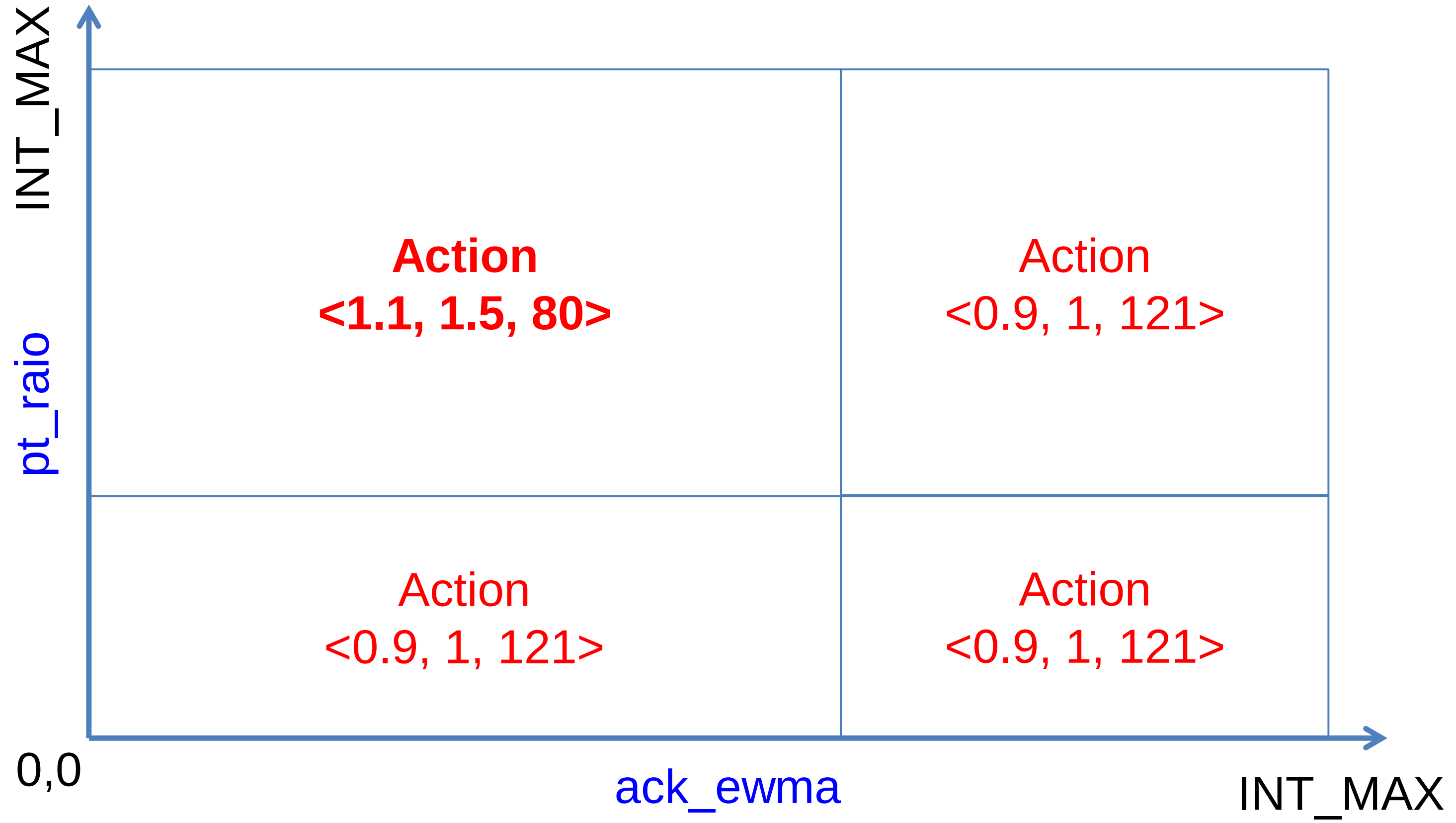
Run the workload, find out the rules that was triggered most often



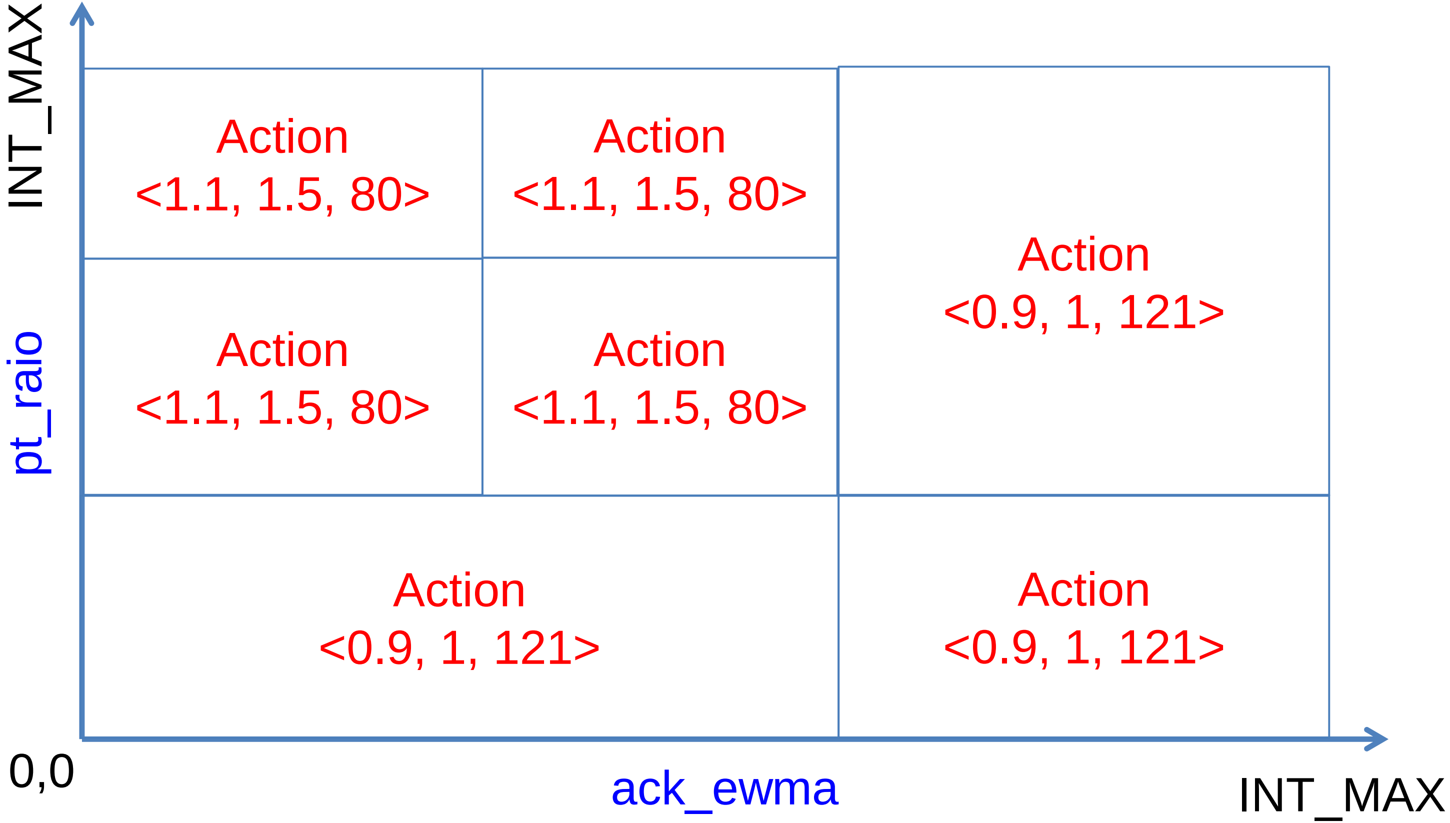
Improve the most used rules by sweeping all possible values of <m, b, τ>



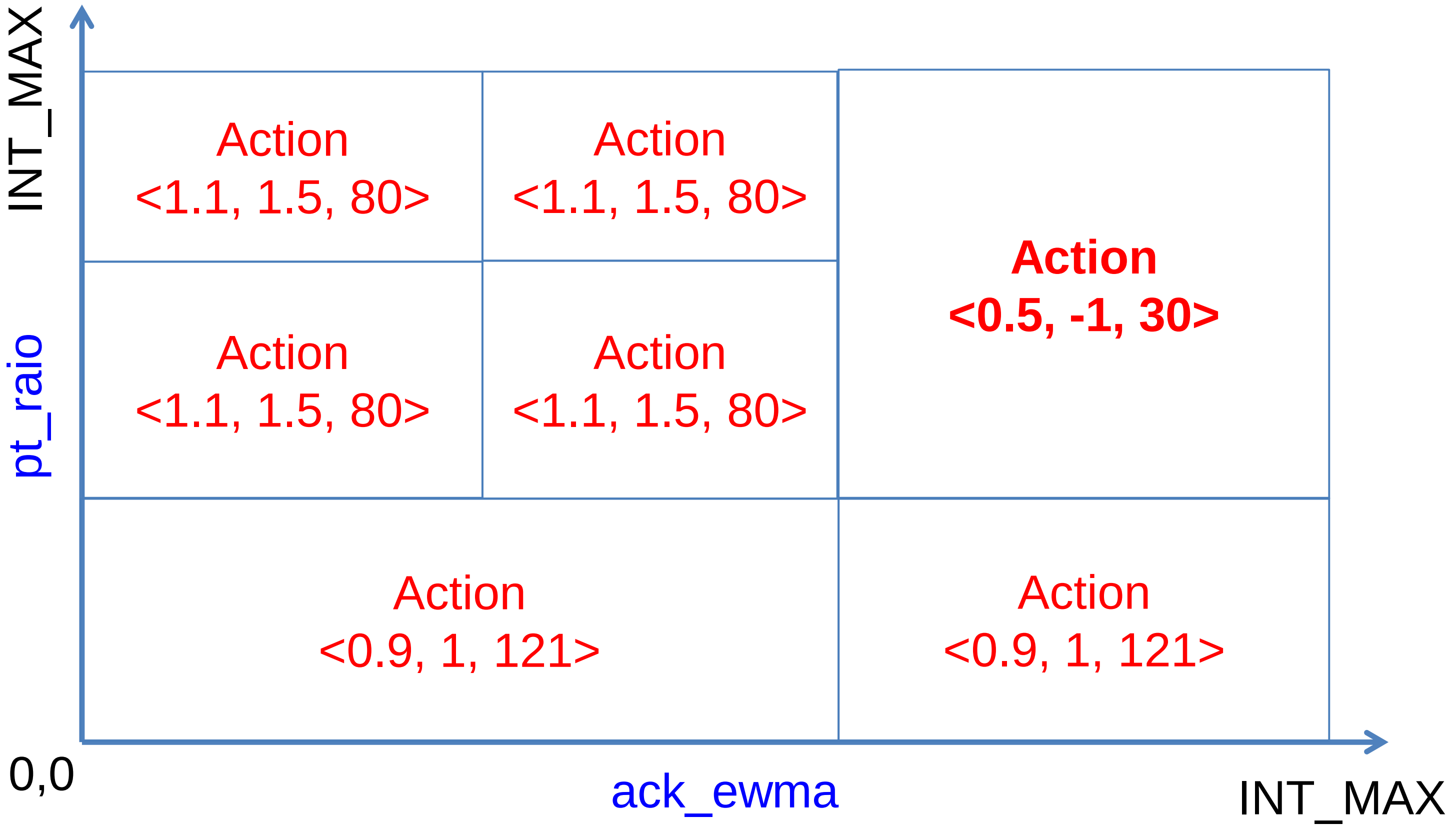
Find the action that works best



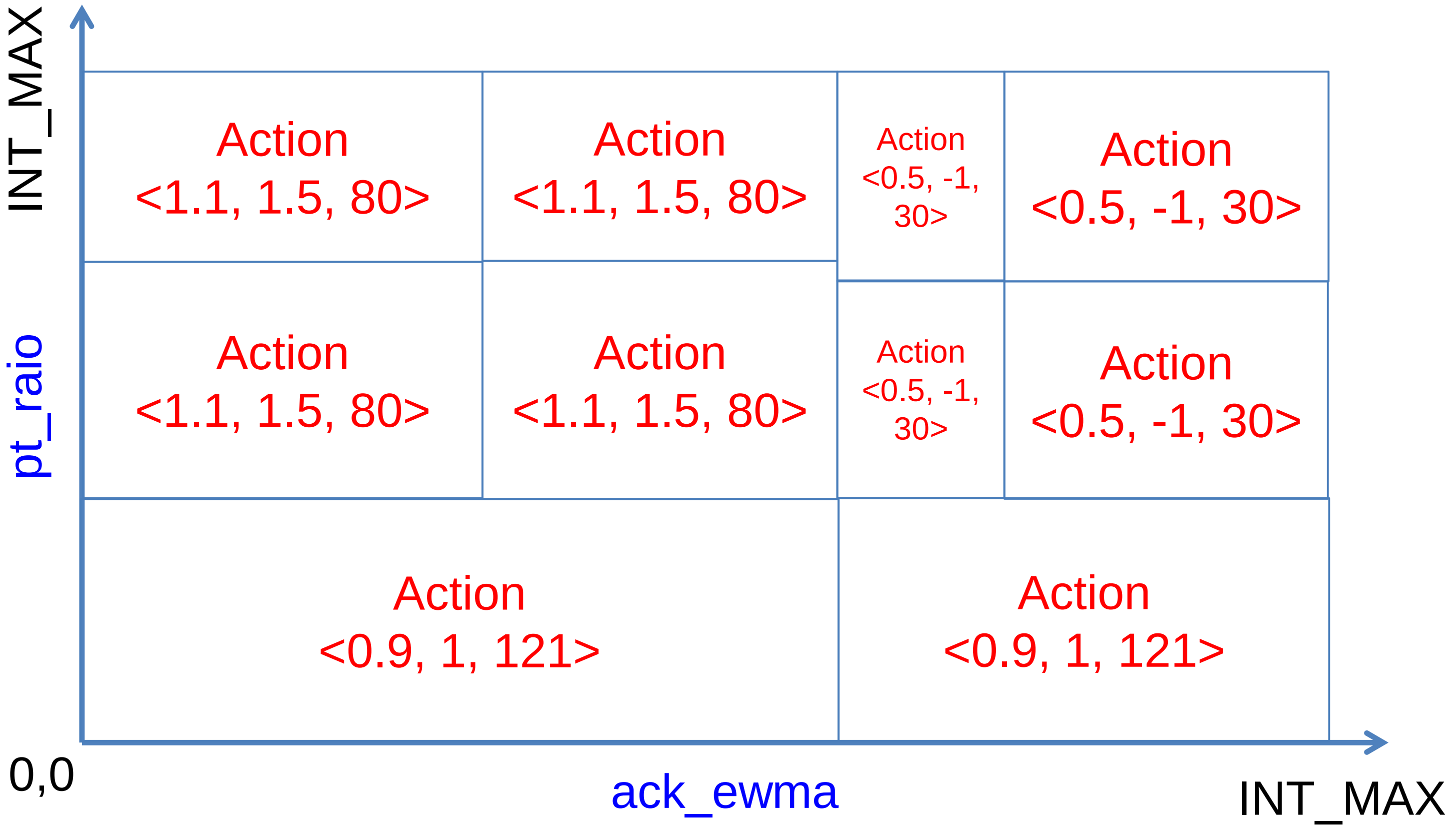
Split the most used rule's state space at the most observed state values



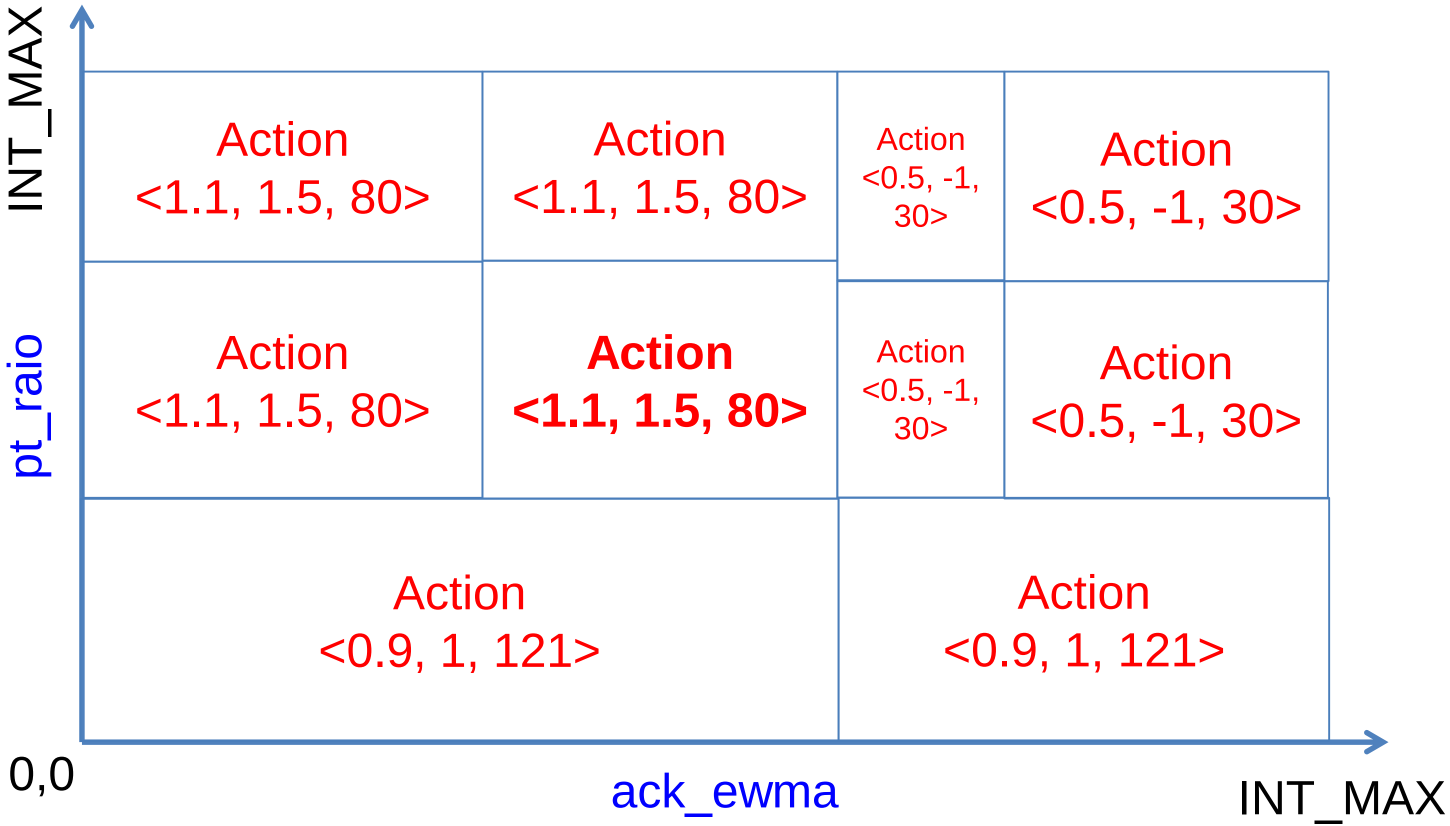
Run workload, find the most used rule, and improve it



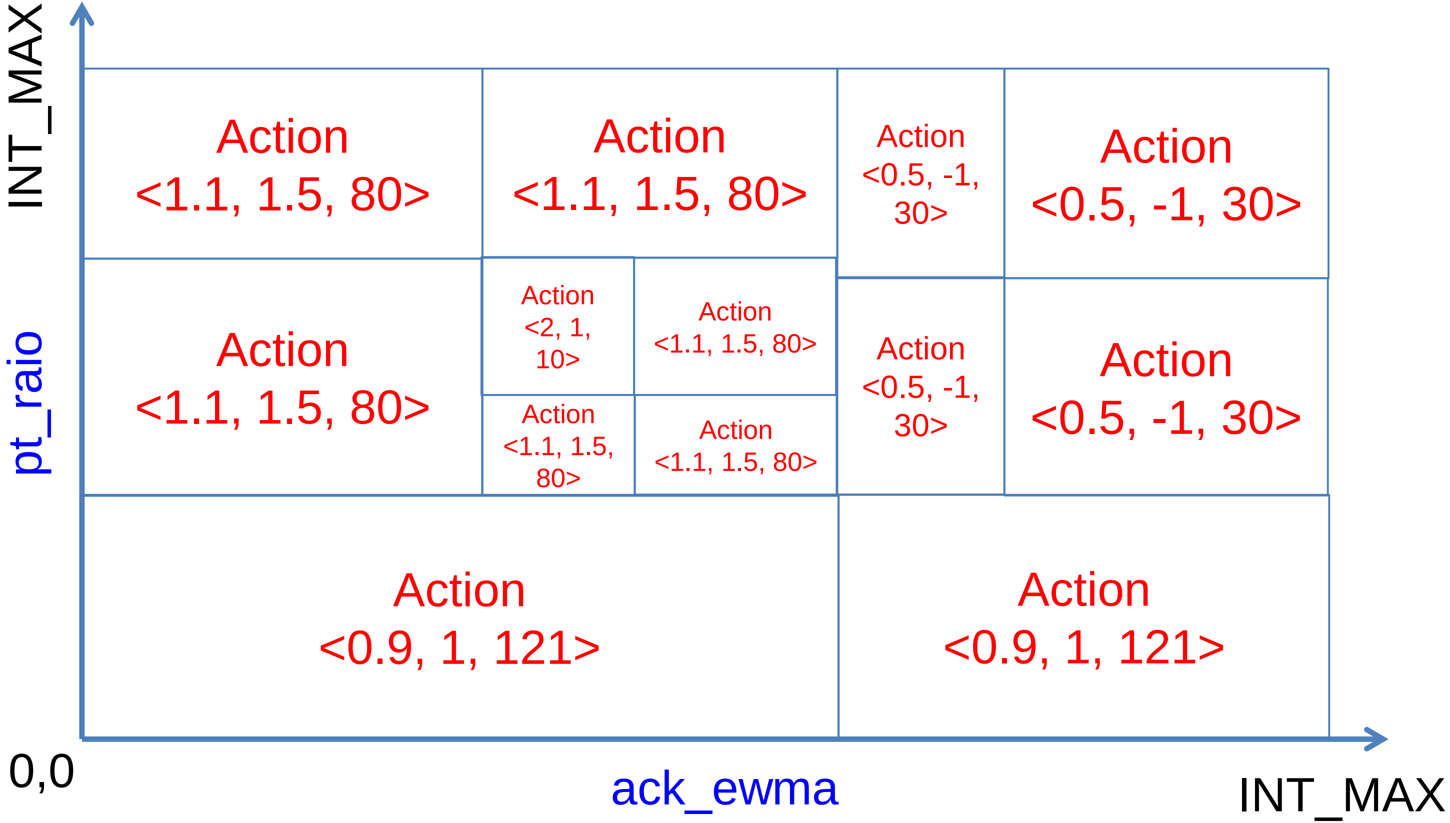
After find the best action, split the most used rule



After find the best action, split the most used rule



Repeat this process



Prototype and Evaluation

An ASCAR prototype for Lustre

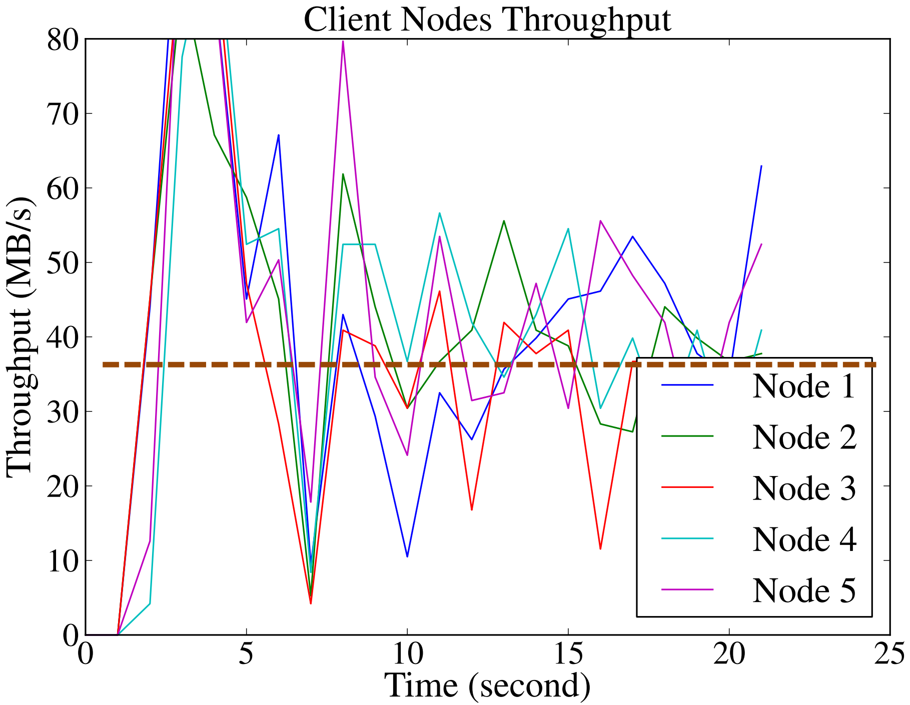
Patched Lustre client to add congestion control

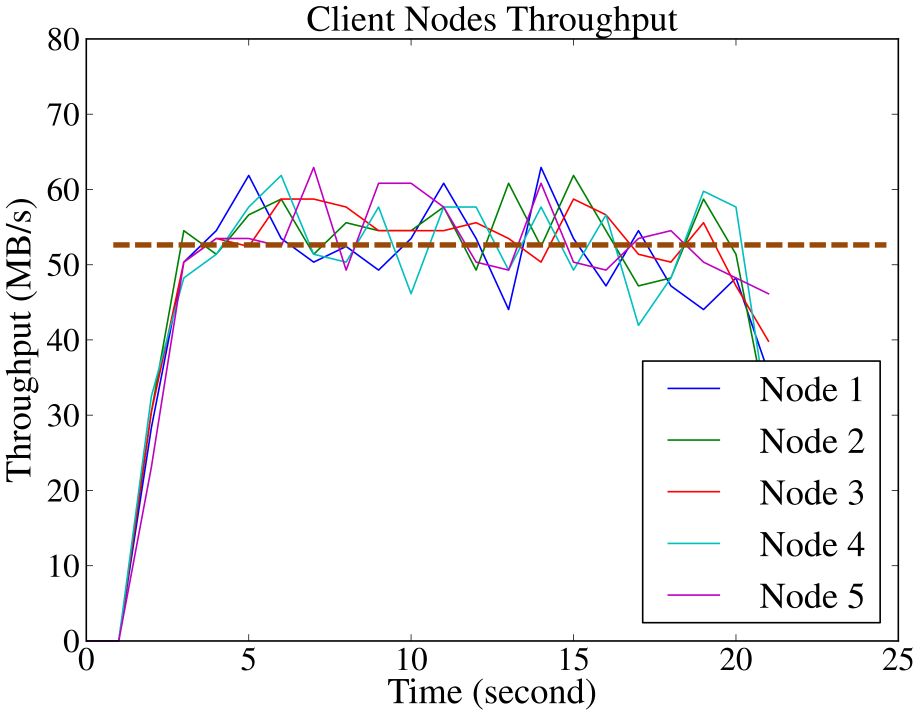
no change to server or other parts

Hardware: 5 servers, 5 clients

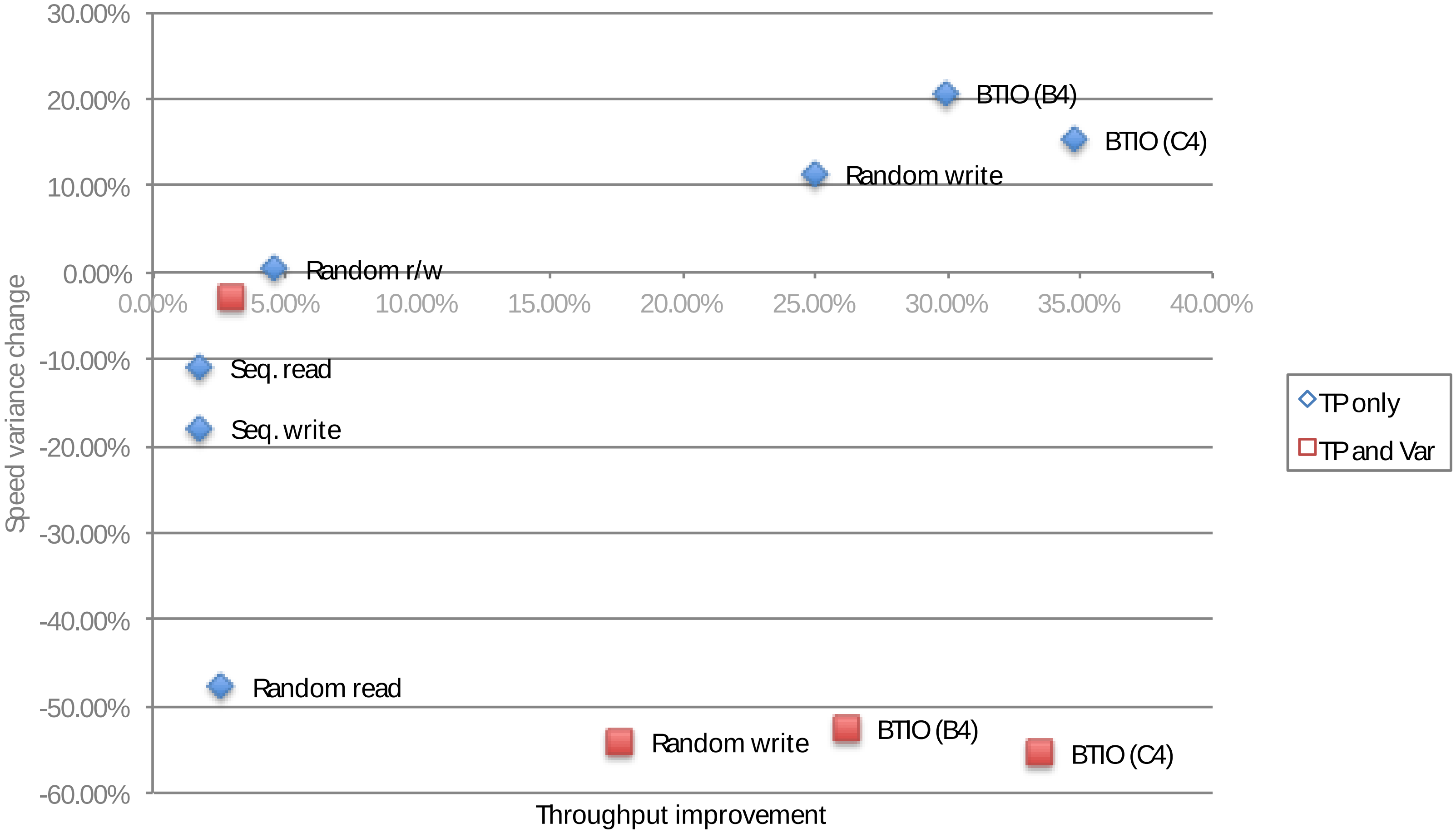
Intel Xeon CPU E3-1230 V2 @ 3.30GHz, 16 GB RAM,
Intel 330 SSD for the OS,
dedicated 7200 RPM HGST Travelstar Z7K500 hard drive for Lustre,
Gigabit Ethernet

ASCAR is good at increasing throughout and decreasing speed variance

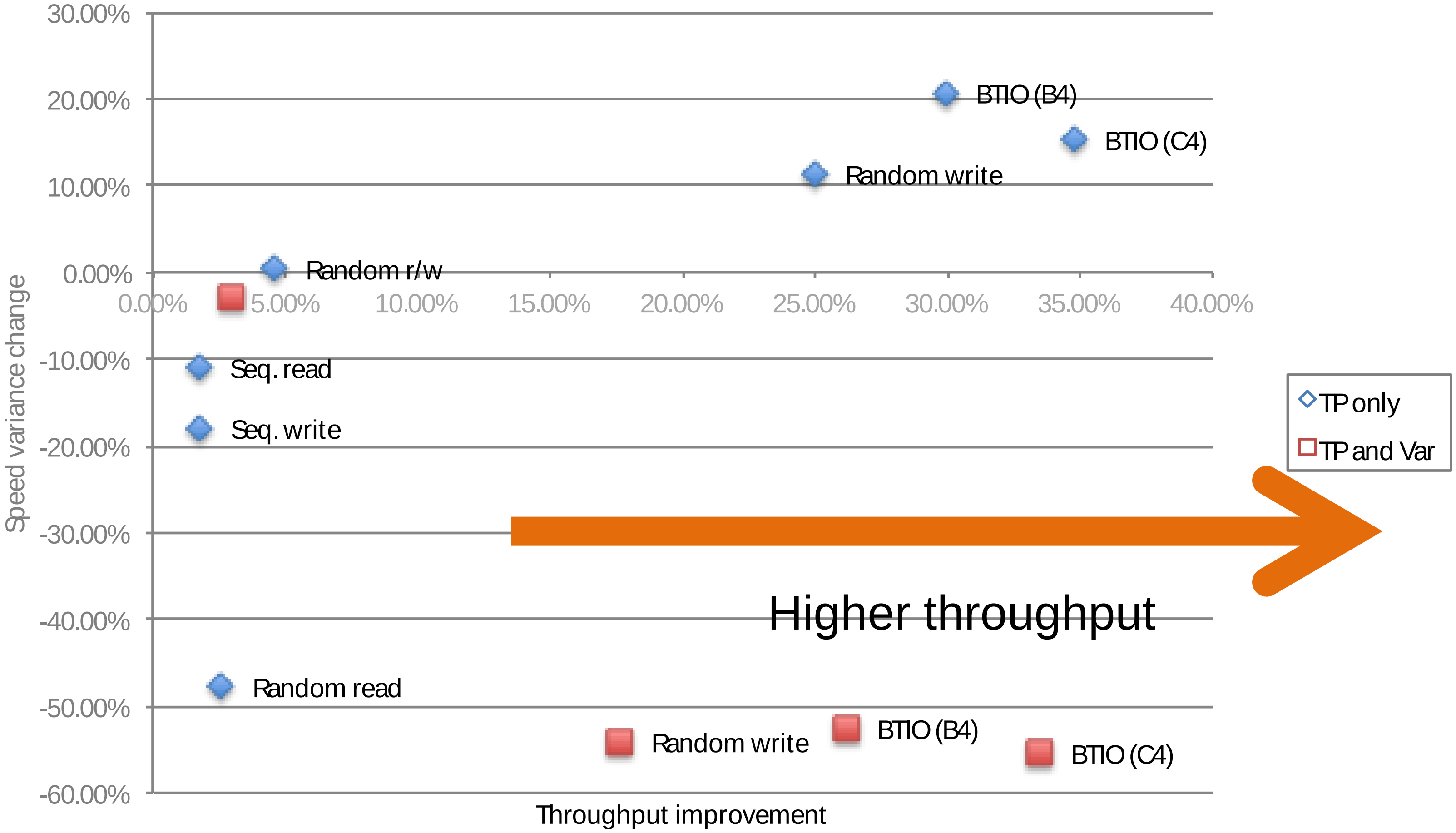




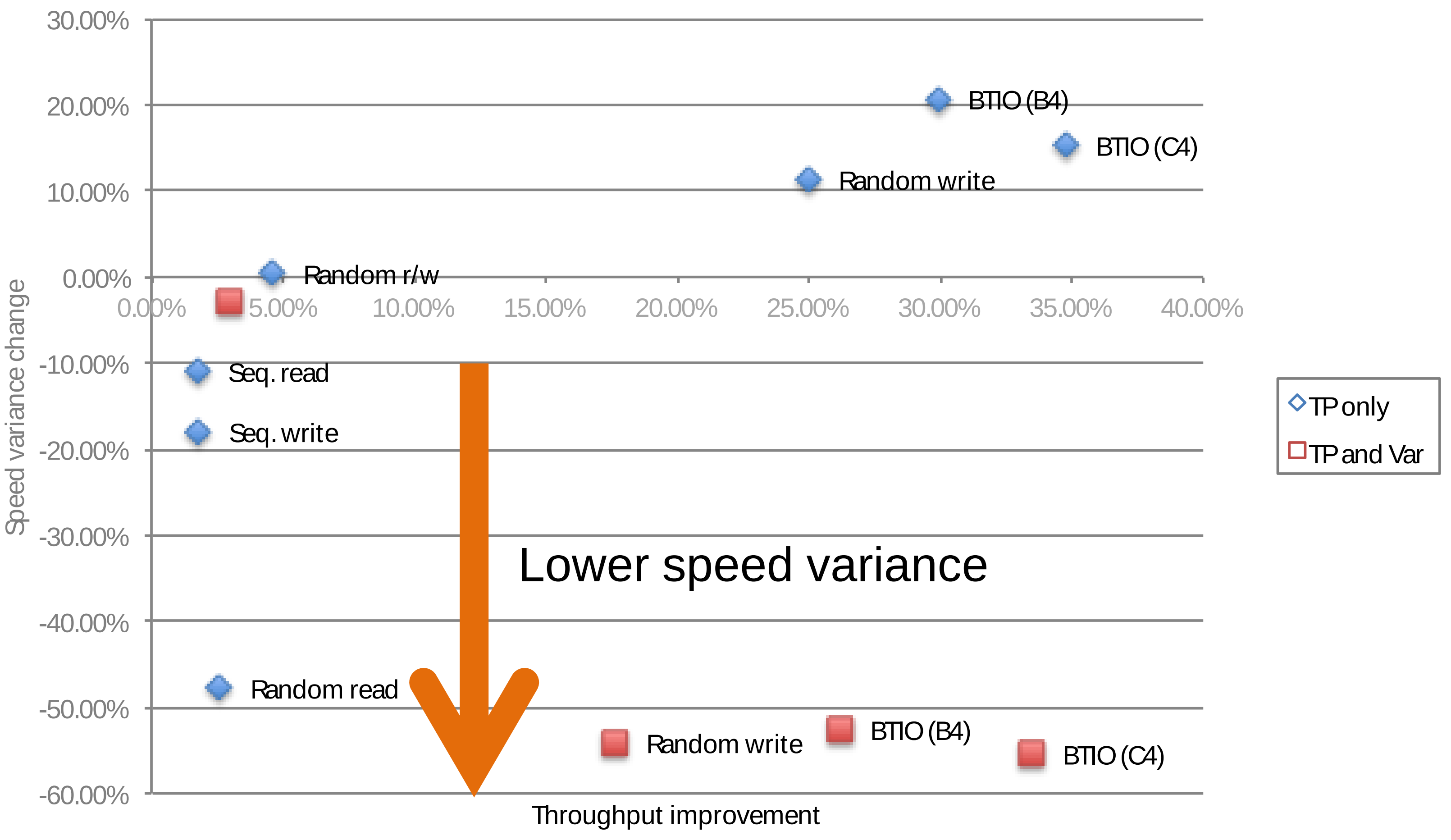
Workload Throughput Improvements



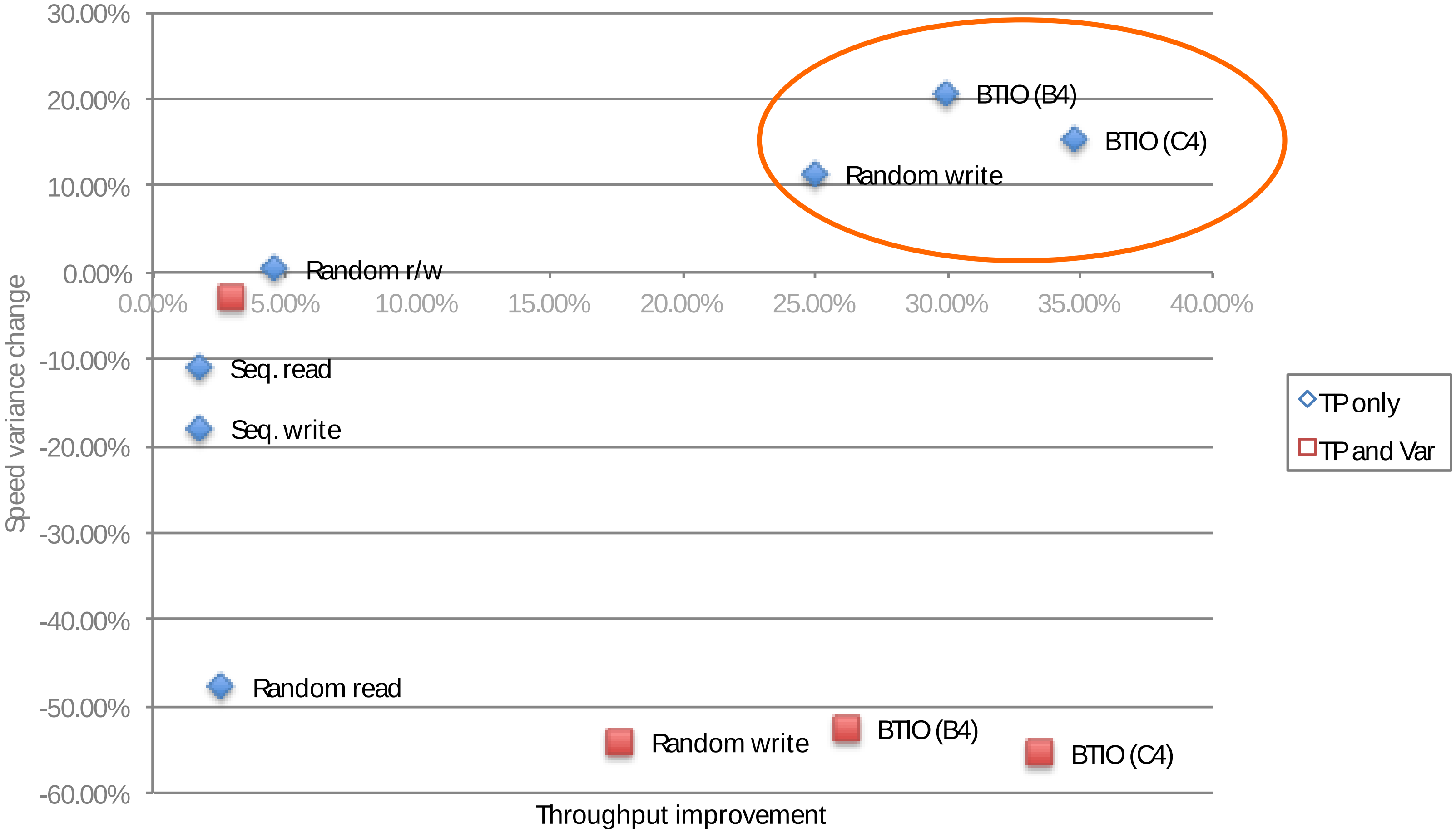
Workload Throughput Improvements



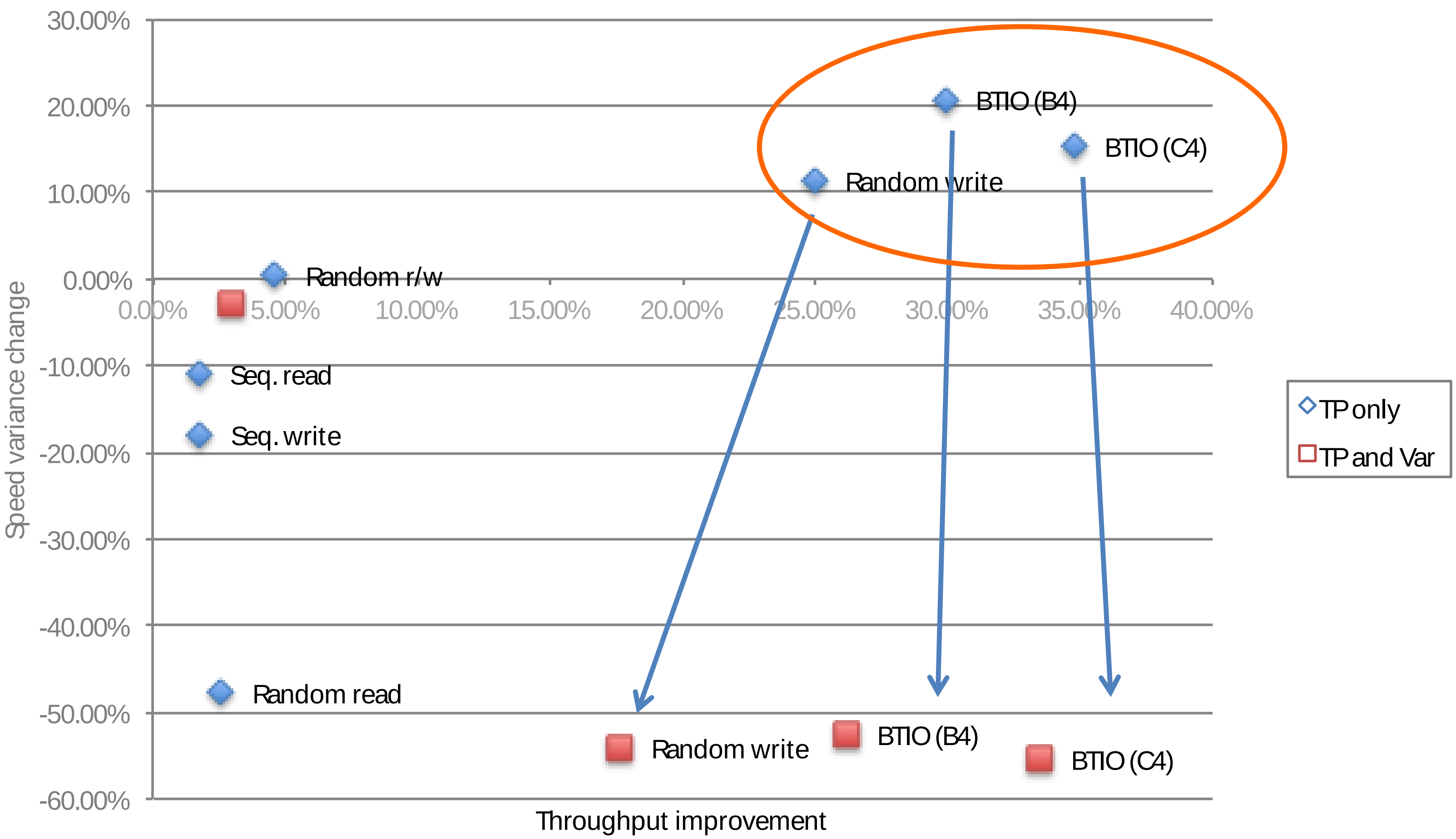
Workload Throughput Improvements



Workload Throughput Improvements



Workload Throughput Improvements



Our prototype shows that ...

ASCAR increases the performance of all workloads

2% to 36%

ASCAR works best to boost write-heavy workloads

25% to 36%

ASCAR can lower speed variance at the same time for certain workloads

by more than 50%

ascar.io

ssrc

Limitations

Training is currently offline

The offline training can take hours

Need to re-train when workload changes

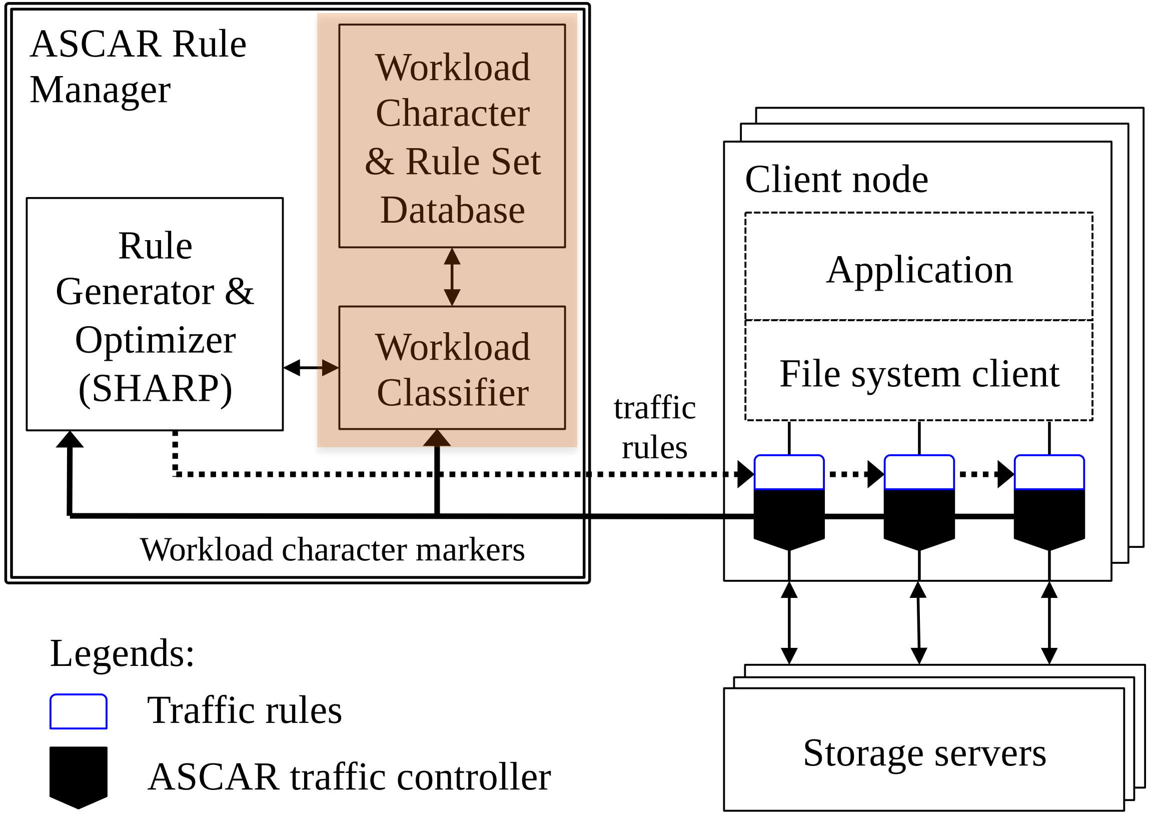
Handling a new workload without training

Measure the workload’s features

Compare features with known workloads

Find the most similar known workload and use its contention control rules

Components of the ASCAR prototype



Finding the right features

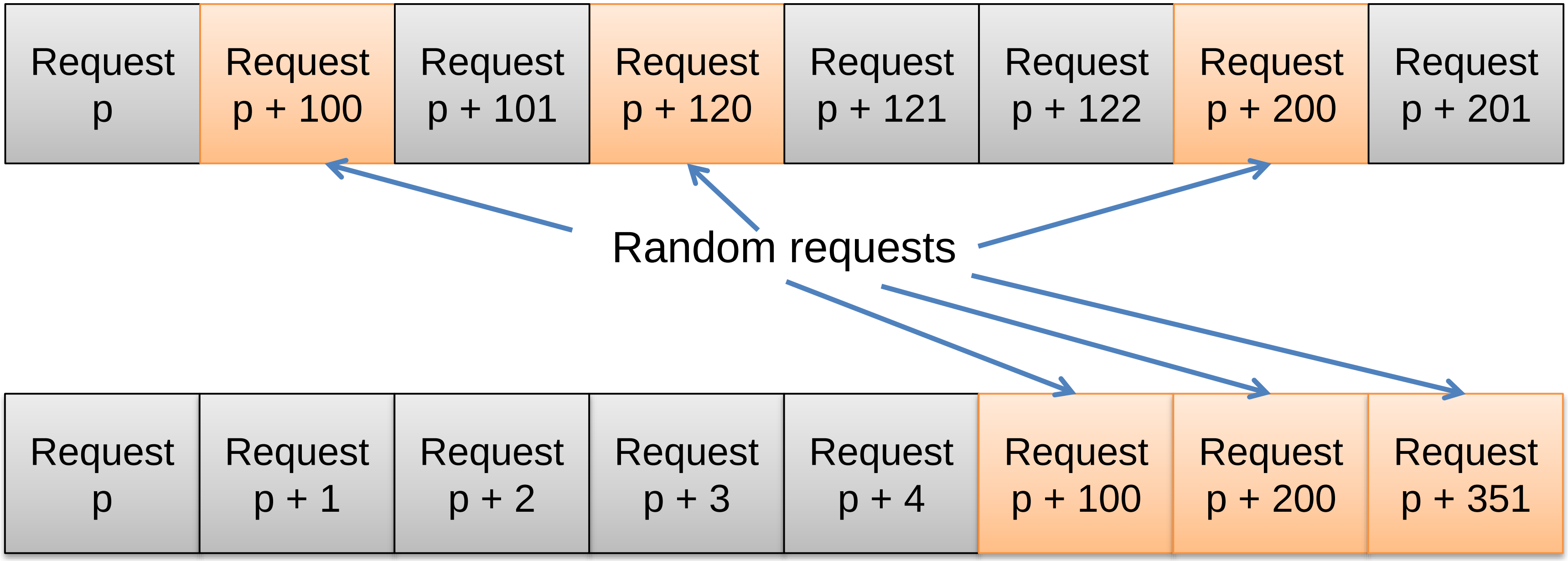
Workloads can have radically different *pressure* on the underlying system

they require different congestion control rules

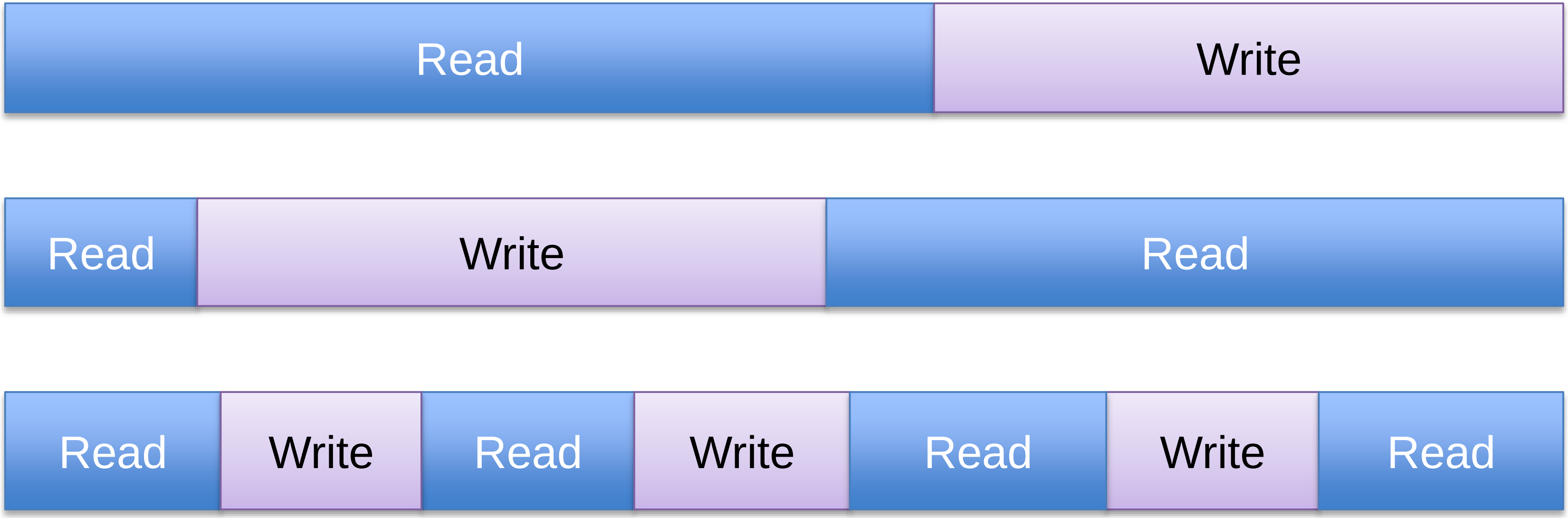
The features should reflect the workload's *pressure* on the underlying system

The combined workload from many clients is a mix of read/write and random/sequential

A 75% sequential + 25% random workload can be very different from another



And there are many different 60% read + 40% write workloads out there



The feature set we use

Op type: read/write/metadata

Ratios between ops (read to write, read/write to metadata, etc.)

For each type of op, we measure the following features:
1. average size of sequential ops
2. average positional gap between seq. ops
3. average temporal gap between seq. ops

Sample:
Different 60% read + 40% write workloads

Read to write: 60/40
Avg. size of sequential read: 60 MB
Avg. size of sequential write: 40 MB

Read to write: 60/40
Avg. size of sequential read: 15 MB
Avg. size of sequential write: 13 MB

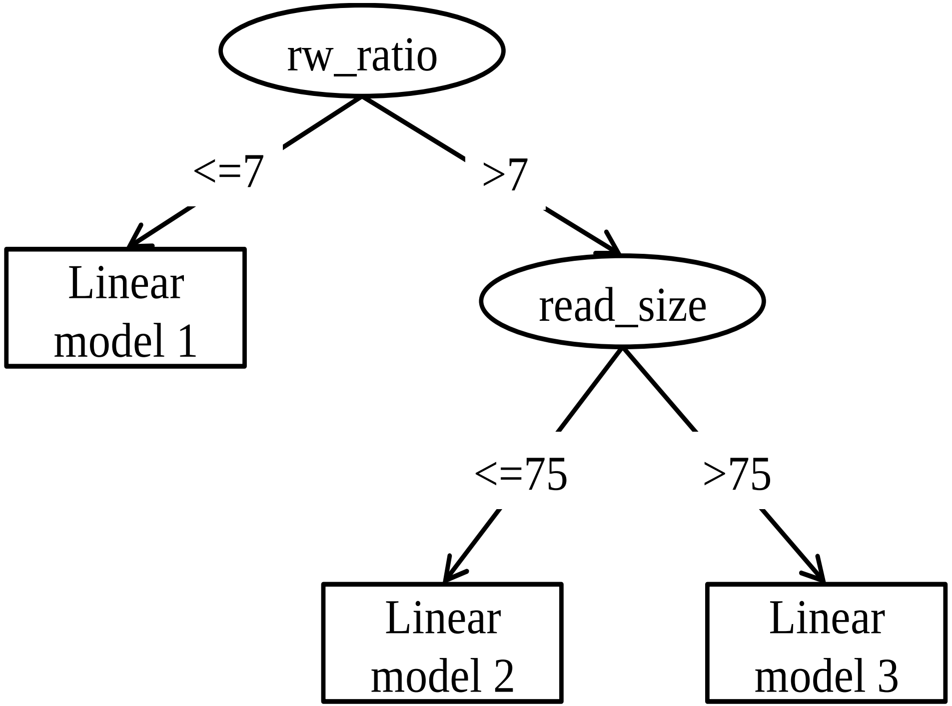
Calculating similarity of workloads

Goal: to determine if they can use the same contention control rules

How: start from a known workload and tweak its features, measuring the efficiency of existing rules on the changed workload

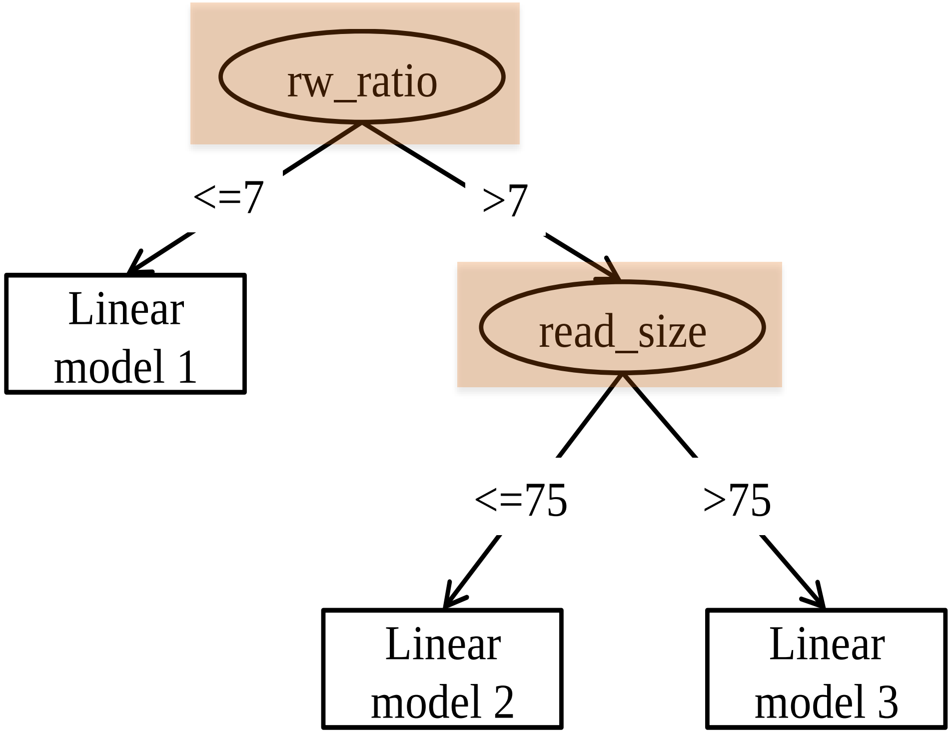
Result: we used the results of hundreds of benchmarks to generate a decision tree

Calculating similarity of workloads



ascar.io ssrc

Calculating similarity of workloads



This decision tree can precisely predict the performance of using a specific rule set with a new workload

Correlation coefficient: 0.8676

Mean absolute error: 0.038

Root mean squared error: 0.0462

Not a Conclusion: ASCAR …

Does not require knowledge of the system or workloads
fully unsupervised

Only needs client-side controllers
no need to change hardware/application/network/server software

Can improve highly changeable workloads
like burst I/O

Work-conserving
no wasted bandwidth

ascar.io
ssrc

Not a Conclusion: ASCAR …

Needs to be evaluated on a larger scale

we are looking for collaborators

Can be evaluated using a patched client (2.4, 2.7)

no need to change server or hardware

Not a Conclusion: ASCAR ...

Paper and source code of our prototype are published

@ http://ascar.io

Future work: online rule optimization

Current ASCAR prototype requires a lengthy offline learning process

Use cloud computing services for doing evaluation on a larger scale

Online tweaking of rules using random-restart hill climbing

Also need to evaluate the ASCAR algorithm on other workloads: database, web services

Acknowledgments

This research was supported in part by the National Science Foundation under awards IIP-1266400, CCF-1219163, CNS- 1018928, the Department of Energy under award DE-FC02- 10ER26017/DESC0005417, Symantec Graduate Fellowship, and industrial members of the Center for Research in Storage Systems. We would like to thank the sponsors of the Storage Systems Research Center (SSRC), including Avago Technologies, Center for Information Technology Research in the Interest of Society (CITRIS of UC Santa Cruz), Department of Energy/Office of Science, EMC, Hewlett Packard Laboratories, Intel Corporation, National Science Foundation, NetApp, Sandisk, Seagate Technology, Symantec, and Toshiba for their generous support.

ASCAR project: http://ascar.io

Contact:
Yan Li <yanli@cs.ucsc.edu>

ascar.io
ssrc