Targeted Resource Management in Multi-tenant Distributed Systems

Jonathan Mace
Brown University

Peter Bodik
MSR Redmond

Rodrigo Fonseca
Brown University

Madanlal Musuvathi
MSR Redmond

Retro
Resource Management in Multi-Tenant Systems
Failure in availability zone cascades to shared control plane, causes thread pool starvation for all zones

- April 2011 – Amazon EBS Failure
Resource Management in Multi-Tenant Systems

Failure in availability zone **cascades to shared control plane**, causes thread pool starvation for all zones
- April 2011 – Amazon EBS Failure

Aggressive **background task** responds to increased hardware capacity with deluge of warnings and logging
- Aug. 2012 – Azure Storage Outage
Resource Management in Multi-Tenant Systems

- Failure in availability zone **cascades to shared control plane**, causes thread pool starvation for all zones
  - April 2011 – Amazon EBS Failure

- Aggressive **background task** responds to increased hardware capacity with deluge of warnings and logging
  - Aug. 2012 – Azure Storage Outage

- Code change increases database usage, shifts bottleneck to unmanaged **application-level lock**
  - Nov. 2014 – Visual Studio Online outage
Failure in availability zone **cascades to shared control plane**, causes thread pool starvation for all zones
- April 2011 – Amazon EBS Failure

Aggressive **background task** responds to increased hardware capacity with deluge of warnings and logging
- Aug. 2012 – Azure Storage Outage

Code change increases database usage, shifts bottleneck to unmanaged **application-level lock**
- Nov. 2014 – Visual Studio Online outage

**Shared storage layer** bottlenecks circumvent resource management layer
- 2014 – Communication with Cloudera
Resource Management in Multi-Tenant Systems

Failure in availability zone **cascades to shared control plane**, causes thread pool starvation for all zones
• April 2011 – Amazon EBS Failure

Aggressive **background task** responds to increased hardware capacity with deluge of warnings and logging
• Aug. 2012 – Azure Storage Outage

Code change increases database usage, shifts bottleneck to unmanaged **application-level lock**
• Nov. 2014 – Visual Studio Online outage

**Shared storage layer** bottlenecks circumvent resource management layer
• 2014 – Communication with Cloudera

Degraded performance, Violated SLOs, system outages
Resource Management in Multi-Tenant Systems

- Degraded performance, Violated SLOs, system outages

Failure in availability zone **cascades to shared control plane**, causes thread pool starvation for all zones
- April 2011 – Amazon EBS Failure

Aggressive **background task** responds to increased hardware capacity with deluge of warnings and logging
- Azure Storage Outage

.resource Management layer** bottlenecks circumvent resource management layer
- 2014 – Communication with Cloudera
Resource Management in Multi-Tenant Systems

Failure in availability zone cascades to shared control plane, causes thread pool starvation for all zones
- April 2011 - Amazon EBS Failure

Aggressive background task responds to increased hardware capacity with deluge of warnings and logging
- Aug. 2012 - Azure Storage Outage

Increases database usage, shifts unmanaged application-level lock
- Visual Studio Online outage

Shared storage layer bottlenecks circumvent resource management layer
- 2014 - Communication with Cloudera

Degraded performance, Violated SLOs, system outages
Resource Management in Multi-Tenant Systems

Failure in availability zone causes thread pool starvation for all zones

- April 2011 - Amazon EBS Failure

Aggressive background task increases database usage, shifts bottleneck to unmanaged application

- Aug. 2012 - Azure Storage Outage

Shared storage layer bottlenecks circumvent resource management layer

- 2014 - Communication with Cloudera

Degraded performance, Violated SLOs, system outages
Containers / VMs
Containers / VMs

Shared Systems:
Storage, Database, Queueing, etc.
Monitors resource usage of each tenant in near real-time.
Actively schedules tenants and activities.
Monitors resource usage of each tenant in near real-time
Actively schedules tenants and activities

High-level, centralized policies:
Encapsulates resource management logic
Monitors resource usage of each tenant in near real-time
Actively schedules tenants and activities

High-level, centralized policies:
  Encapsulates resource management logic
  Abstractions – not specific to resource type, system
  Achieve different goals: guarantee average latencies, fair share a resource, etc.
Hadoop Distributed File System (HDFS)
Hadoop Distributed File System (HDFS)

HDFS NameNode

Filesystem metadata
Hadoop Distributed File System (HDFS)
Hadoop Distributed File System (HDFS)
Hadoop Distributed File System (HDFS)

- **HDFS NameNode**
  - Filesystem metadata

- **HDFS DataNode**
  - Replicated block storage

**Actions:**
- Rename
- Read
Hadoop Distributed File System (HDFS)

Filesystem metadata

Replicated block storage

Read

Rename
random 8kb reads

HDFS NameNode

HDFS DataNode

HDFS DataNode

HDFS DataNode
random 8kb reads
random 4Mb reads

request latency [ms]
disk op latency [ms]
HDFS NameNode

HDFS DataNode

HDFS DataNode

HDFS DataNode

random 8kb reads

random 4Mb reads

list directory

rename files

request latency [ms]

disk op latency [ms]

queue latency [ms]

lock latency [ms]
HDFS DataNode

Local storage
Local storage

HDFS DataNode
Retro
Goals
Retro

Goals

Co-ordinated control across processes, machines, and services
Co-ordinated control across processes, machines, and services

Handle system and application level resources
Retro

Goals

Co-ordinated control across processes, machines, and services

Handle system and application level resources

Principals: tenants, background tasks
Co-ordinated control across processes, machines, and services
Handle system and application level resources
Principals: tenants, background tasks
Real-time and reactive
Co-ordinated control across processes, machines, and services
Handle system and application level resources
Principals: tenants, background tasks
Real-time and reactive

Efficient: Only control what is needed
Retro Architecture
Tenant Requests
Workflows

Purpose: identify requests from different users, background activities

eg, all requests from a tenant over time

eg, data balancing in HDFS
Workflows

Purpose: identify requests from different users, background activities
eg, all requests from a tenant over time
eg, data balancing in HDFS

Unit of resource measurement, attribution, and enforcement
Workflows

Purpose: identify requests from different users, background activities

eg, all requests from a tenant over time
eg, data balancing in HDFS

Unit of resource measurement, attribution, and enforcement

Tracks a request across varying levels of granularity

Orthogonal to threads, processes, network flows, etc.
Workflows
Workflows
Workflows

Purpose: cope with diversity of resources
Purpose: cope with diversity of resources

What we need:

1. Identify overloaded resources
Purpose: cope with diversity of resources

What we need:

1. Identify overloaded resources

2. Identify culprit workflows
Purpose: cope with diversity of resources

What we need:

1. Identify overloaded resources

   **Slowdown**  
   Ratio of how slow the resource is now compared to its baseline performance with no contention.

2. Identify culprit workflows
Purpose: cope with diversity of resources

What we need:

1. Identify overloaded resources
   - **Slowdown** Ratio of how slow the resource is now compared to its baseline performance with no contention.

2. Identify culprit workflows
   - **Load** Fraction of current utilization that we can attribute to each workflow
Purpose: cope with diversity of resources

What we need:

1. Identify overloaded resources

   **Slowdown**  
   Ratio of how slow the resource is now compared to its baseline performance with no contention.

2. Identify culprit workflows

   **Load**  
   Fraction of current utilization that we can attribute to each workflow
Purpose: cope with diversity of resources

What we need:

1. Identify overloaded resources
   - **Slowdown**: Ratio of how slow the resource is now compared to its baseline performance with no contention.

2. Identify culprit workflows
   - **Load**: Fraction of current utilization that we can attribute to each workflow
Purpose: cope with diversity of resources

What we need:

1. Identify overloaded resources

   **Slowdown**
   Ratio of how slow the resource is now compared to its baseline performance with no contention.

   (queue time + execute time) / execute time
   eg. 100ms queue, 10ms execute
   => slowdown 11

2. Identify culprit workflows

   **Load**
   Fraction of current utilization that we can attribute to each workflow
   time spent executing
   eg. 10ms execute
   => load 10
Goal: enforce resource management decisions
Goal: enforce resource management decisions
Decoupled from resources
Rate-limits workflows, agnostic to underlying implementation e.g.,
token bucket
priority queue
Goal: enforce resource management decisions
Decoupled from resources
Rate-limits workflows, agnostic to underlying implementation e.g.,
token bucket
priority queue
Goal: enforce resource management decisions
Decoupled from resources
Rate-limits workflows, agnostic to underlying implementation e.g.,
  token bucket
  priority queue
Goal: enforce resource management decisions
Decoupled from resources
Rate-limits workflows, agnostic to underlying implementation e.g.,
token bucket
priority queue
Goal: enforce resource management decisions
Decoupled from resources
Rate-limits workflows, agnostic to underlying implementation e.g.,
token bucket
priority queue
Goal: enforce resource management decisions
Decoupled from resources
Rate-limits workflows, agnostic to underlying implementation e.g.,
token bucket
priority queue
Goal: enforce resource management decisions
Decoupled from resources
Rate-limits workflows, agnostic to underlying implementation e.g.,
token bucket
priority queue
Goal: enforce resource management decisions
Decoupled from resources
Rate-limits workflows, agnostic to underlying implementation e.g.,
token bucket
priority queue
Workflows

Resources

Control Points
1. Pervasive Measurement
Aggregated locally then reported centrally once per second
1. Pervasive Measurement
   Aggregated locally then reported centrally once per second

2. Centralized Controller
   Global, abstracted view of the system
1. Pervasive Measurement
Aggregated locally then reported centrally once per second

2. Centralized Controller
Global, abstracted view of the system
Policies run in continuous control loop
1. **Pervasive Measurement**
   Aggregated locally then reported centrally once per second

2. **Centralized Controller**
   Global, abstracted view of the system
   Policies run in continuous control loop

3. **Distributed Enforcement**
   Co-ordinates enforcement using distributed token bucket
“Control Plane” for resource management
Global, abstracted view of the system
Easier to write
Reusable
Example: LatencySLO
Example: Latency SLO

H High Priority Workflows

“200ms average request latency”
Example: Latency SLO

H  High Priority Workflows

L  Low Priority Workflows

“200ms average request latency”

(use spare capacity)
Example: Latency SLO

H  High Priority Workflows

L  Low Priority Workflows

“200ms average request latency”

monitor latencies

(use spare capacity)
Example: Latency SLO

H  High Priority Workflows

L  Low Priority Workflows

“200ms average request latency”

monitor latencies

attribute interference

(use spare capacity)
Example: Latency SLO

- **H** High Priority Workflows
- **L** Low Priority Workflows

“200ms average request latency”

- Monitor latencies
- Throttle interfering workflows
- (use spare capacity)
foreach candidate in H
    miss[candidate] = \texttt{latency}(candidate) / \texttt{guarantee}[candidate]
W = \texttt{candidate in H with max missing}[candidate]

foreach rsrch in resources() // calculate importance of each resource for hipri
    \texttt{importance}[rsrch] = \texttt{latency}(W, rsrch) \times \log(\texttt{slowdown}(rsrch))

foreach lopri in L // calculate low priority workflow interference
    \texttt{interference}[lopri] = \Sigma_{rsrch} \texttt{importance}[rsrch] \times \texttt{load}(lopri, rsrch) / \texttt{load}(rsrch)

foreach lopri in L // normalize interference
    \texttt{interference}[lopri] /= \Sigma_{k} \texttt{interference}[k]

foreach lopri in L
    if \texttt{miss}[W] > 1 // throttle
        scalefactor = 1 - \alpha \times (\texttt{miss}[W] - 1) \times \texttt{interference}[lopri]
    else // release
        scalefactor = 1 + \beta

foreach cpoint in controlpoints() // apply new rates
    \texttt{set_rate}(cpoint, lopri, scalefactor \times \texttt{get_rate}(cpoint, lopri)
**Policy**

Example: Latency SLO

H  High Priority Workflows  L  Low Priority Workflows

1. `foreach candidate in H`
2. `miss[candidate] = latency(candidate) / guarantee[candidate]`
3. `W = candidate in H with max miss[candidate]`

4. `foreach rsrc in resources()`  // calculate importance of each resource for hipri
5. `importance[rsrc] = latency(W, rsrc) * log(slowdown(rsrc))`

6. `foreach lopri in L`  // calculate low priority workflow interference
7. `interference[lopri] = \sum_{rsr} importance[rsr] * load(lopri, rsr) / load(rsrc)`

8. `foreach lopri in L`  // normalize interference
9. `interference[lopri] /= \sum_k interference[k]`

10. `foreach lopri in L`
11.   `if miss[W] > 1  // throttle`
12.     `scalefactor = 1 - \alpha * (miss[W] - 1) * interference[lopri]`
13.   `else  // release`
14.     `scalefactor = 1 + \beta`

15. `foreach cpoint in controlpoints()`  // apply new rates
16.   `set_rate(cpoint, lopri, scalefactor * get_rate(cpoint, lopri)}`
**Policy**

**Example: LatencySLO**

Select the high priority workflow $W$ with worst performance

```python
1. foreach candidate in H
2. miss[candidate] = latency[candidate] / guarantee[candidate]
3. W = candidate in H with max miss[candidate]

4. foreach rsrc in resources() // calculate importance of each resource for hipri
5. importance[rsrc] = latency(W, rsrc) * log(slowdown(rsrsr))

6. foreach lopri in L // calculate low priority workflow interference
7. interference[lopri] = Σrsrsrc importance[rsrsrc] * load(lopri, rsrsrc) / load(rsrsrc)

8. foreach lopri in L // normalize interference
9. interference[lopri] /= Σk interference[k]

10. foreach lopri in L
11. if miss[W] > 1 // throttle
12. scalefactor = 1 - α * (miss[W] - 1) * interference[lopri]
13. else // release
14. scalefactor = 1 + β

15. foreach cpoint in controlpoints() // apply new rates
16. set_rate(cpoint, lopri, scalefactor * get_rate(cpoint, lopri))
```

**Example:** Latency SLO

H High Priority Workflows

L Low Priority Workflows

Select the high priority workflow $W$ with worst performance.
Select the high priority workflow $W$ with worst performance

Weight low priority workflows by their interference with $W$

Example: LatencySLO

<table>
<thead>
<tr>
<th>Policy</th>
<th>Example: LatencySLO</th>
</tr>
</thead>
<tbody>
<tr>
<td>H</td>
<td>High Priority Workflows</td>
</tr>
<tr>
<td>L</td>
<td>Low Priority Workflows</td>
</tr>
</tbody>
</table>

1. foreach candidate in H

2. $W = \text{candidate in } H \text{ with max miss[candidate]}$

3. foreach rsrc in resources() // calculate importance of each resource for hipri

4. $\text{importance[rsrc]} = \text{latency}(W, rsrc) \times \log(\text{slowdown}(rsrc))$

5. foreach lopri in L // calculate low priority workflow interference

6. $\text{interference[lopri]} = \Sigma_{rsrc} \text{importance[rsrc]} \times \frac{\text{load(lopri, rsr)}}{\text{load(rsrc)}}$

7. foreach lopri in L // normalize interference

8. $\text{interference[lopri]} /= \Sigma_k \text{interference[k]}$

9. foreach lopri in L // throttle

10. if miss$[W] > 1$

11. $\text{scalefactor} = 1 - \alpha \times (\text{miss}[W] - 1) \times \text{interference[lopri]}$

12. else // release

13. $\text{scalefactor} = 1 + \beta$

14. foreach cpoint in controlpoints() // apply new rates

15. $\text{set_rate}(cpoint, lopri, \text{scalefactor} \times \text{get_rate}(cpoint, lopri)$
**Policy**

Example: LatencySLO

**H** High Priority Workflows  
**L** Low Priority Workflows

1. **foreach** candidate **in** H
   
   **Select the high priority workflow** $W$ **with worst performance**

   2. $W = \text{candidate in } H \text{ with max } \text{miss[candidate]}$

3. **foreach** rsrcc **in** resources()  
   
   **Weight low priority workflows by their interference with** $W$

   4. $\text{importance[rsrcc]} = \text{latency}(W, \text{rsrcc}) \times \log(\text{slowdown}(\text{rsrcc}))$

5. **foreach** lopri **in** L
   
   **Throttle low priority workflows proportionally to their weight**

6. $\text{interference[lopri]} = \sum_{rsrcc} \text{importance[rsrcc]} \times \text{load}(\text{lopri, rsrcc}) / \text{load(rsrc)}$

7. **foreach** lopri **in** L
   
   8. $\text{interference[lopri]} /= \sum_k \text{interference[k]}$

9. **foreach** lopri **in** L
   
   10. $\text{miss[W]} > 1$

   11. $\text{scalefactor} = 1 - \alpha \times (\text{miss[W]} - 1) \times \text{interference[lopri]}$

   12. else

   13. $\text{scalefactor} = 1 + \beta$

14. **foreach** cpoint **in** controlpoints()  
   
   **apply new rates**

15. $\text{set_rate}(\text{cpoint}, \text{lopri}, \text{scalefactor} \times \text{get_rate}(\text{cpoint, lopri})$
Example: LatencySLO

H  High Priority Workflows
L  Low Priority Workflows

Select the high priority workflow \( W \) with worst performance

1. \texttt{foreach candidate in H} \\
2. \hspace{1em} \texttt{miss[candidate] = latency(candidate) / guarantee[candidate]} \\
3. \hspace{1em} \texttt{W = candidate in H with max miss[candidate]} \\

Weight low priority workflows by their interference with \( W \)

4. \texttt{foreach rsr\texttt{c in resources}()} \hspace{1em} // calculate importance of each resource for hipri \\
5. \hspace{2em} \texttt{importance[rsr\texttt{c}]} = \texttt{latency(W, rsr\texttt{c}) * log(slowdown(rsr\texttt{c}))} \\

6. \texttt{foreach lopr\texttt{i in L} } \hspace{1em} // calculate low priority workflow interference \\
7. \hspace{2em} \texttt{interference[lopri]} = \Sigma_{rsr\texttt{c}} \texttt{importance[rsr\texttt{c}]} * \texttt{load(lopri, rsr\texttt{c}) / load(rsr\texttt{c})} \\

8. \texttt{foreach lopri in L} \hspace{1em} // normalize interference \\
9. \hspace{2em} \texttt{interference[lopri]} /= \Sigma_{k} \texttt{interference[k]} \\

10. \texttt{foreach lopri in L} \\
11. \hspace{1em} \texttt{if miss[W] > 1} \hspace{1em} // throttle \\
12. \hspace{2em} \texttt{scalefactor = 1 - \alpha \times (\texttt{miss[W] - 1}) \times \texttt{interference[lopri]}} \\
13. \hspace{1em} \texttt{else} \hspace{1em} // release \\
14. \hspace{2em} \texttt{scalefactor = 1 + \beta} \\

Throttle low priority workflows proportionally to their weight \\

15. \texttt{foreach cpoint in controlpoints()} \hspace{1em} // apply new rates \\
16. \hspace{2em} \texttt{set_rate(cpoint, lopri, scalefactor \times \texttt{get_rate(cpoint, lopri)}}
Policy

Example: LatencySLO

H  High Priority Workflows   L  Low Priority Workflows

Select the high priority workflow $W$ with worst performance

1. \(\text{foreach candidate in } H\)
2. \(\text{miss[candidate]} = \text{latency}(\text{candidate}) / \text{guarantee[candidate]}\)
3. \(W = \text{candidate in } H \text{ with max miss[candidate]}\)

Weight low priority workflows by their interference with $W$

4. \(\text{foreach rsr}c \text{ in resources()} \quad // \text{calculate importance of each resource for hipri}\)
5. \(\text{importance[rsrc]} = \text{latency}(W, \text{rsrc}) \times \log(\text{slowdown(rsrc)})\)
6. \(\text{foreach lopri in } L \quad // \text{calculate low priority workflow interference}\)
7. \(\text{interference[lopri]} = \sum_{rsrc} \text{importance[rsrc]} \times \text{load(lopri, rsrc)} / \text{load(rsrc)}\)

8. \(\text{foreach lopri in } L \quad // \text{normalize interference}\)
9. \(\text{interference[lopri]} /= \sum_k \text{interference[k]}\)

10. \(\text{foreach lopri in } L\)
11. \(\text{if miss[W]} > 1 \quad // \text{throttle}\)
12. \(\text{scalefactor} = 1 - \alpha \times (\text{miss[W]} - 1) \times \text{interference[lopri]}\)
13. \(\text{else} \quad // \text{release}\)
14. \(\text{scalefactor} = 1 + \beta\)

15. \(\text{foreach cpoint in controlpoints()} \quad // \text{apply new rates}\)
16. \(\text{set_rate(cpoint, lopri, scalefactor \times get_rate(cpoint, lopri)}\)

Example:
LatencySLO

H  High Priority Workflows   L  Low Priority Workflows

Select the high priority workflow $W$ with worst performance

1. \(\text{foreach candidate in } H\)
2. \(\text{miss[candidate]} = \text{latency}(\text{candidate}) / \text{guarantee[candidate]}\)
3. \(W = \text{candidate in } H \text{ with max miss[candidate]}\)

Weight low priority workflows by their interference with $W$

4. \(\text{foreach rsr}c \text{ in resources()} \quad // \text{calculate importance of each resource for hipri}\)
5. \(\text{importance[rsrc]} = \text{latency}(W, \text{rsrc}) \times \log(\text{slowdown(rsrc)})\)
6. \(\text{foreach lopri in } L \quad // \text{calculate low priority workflow interference}\)
7. \(\text{interference[lopri]} = \sum_{rsrc} \text{importance[rsrc]} \times \text{load(lopri, rsrc)} / \text{load(rsrc)}\)

8. \(\text{foreach lopri in } L \quad // \text{normalize interference}\)
9. \(\text{interference[lopri]} /= \sum_k \text{interference[k]}\)

10. \(\text{foreach lopri in } L\)
11. \(\text{if miss[W]} > 1 \quad // \text{throttle}\)
12. \(\text{scalefactor} = 1 - \alpha \times (\text{miss[W]} - 1) \times \text{interference[lopri]}\)
13. \(\text{else} \quad // \text{release}\)
14. \(\text{scalefactor} = 1 + \beta\)

15. \(\text{foreach cpoint in controlpoints()} \quad // \text{apply new rates}\)
16. \(\text{set_rate(cpoint, lopri, scalefactor \times get_rate(cpoint, lopri)}\)

Throttle low priority workflows proportionally to their weight
Example: LatencySLO

H  High Priority Workflows   L  Low Priority Workflows

Select the high priority workflow $W$ with worst performance

1. $\text{foreach candidate in } H$
2. $\text{miss[candidate] = latency(candidate) / guarantee[candidate]}$
3. $W = \text{candidate in } H \text{ with max miss[candidate]}$

Weight low priority workflows by their interference with $W$

4. $\text{foreach rsrc in resources() // calculate importance of each resource for hipri}$
5. $\text{importance[rsrc] = latency}(W, \text{rsrc}) \times \log(\text{slowdown}(\text{rsrc}))$

6. $\text{foreach lopri in } L$
7. $\text{interference[lopri] = } \Sigma_\text{rsrc} \text{ importance[rsrc] } \times \text{load}(\text{lopri}, \text{rsrc}) / \text{load}(\text{rsrc})$

Throttle low priority workflows proportionally to their weight

8. $\text{foreach lopri in } L$
9. $\text{interference[lopri] } =\Sigma_\text{k} \text{ interference[}\text{k}]$

10. $\text{foreach lopri in } L$
11. $\text{if miss}[W] > 1 \text{ // throttle}$
12. $\text{scalefactor} = 1 - \alpha * (\text{miss}[W] - 1) * \text{interference[lopri]}$
13. $\text{else} \text{ // release}$
14. $\text{scalefactor} = 1 + \beta$

15. $\text{foreach cpoint in controlpoints() // apply new rates}$
16. $\text{set_rate}(\text{cpoint}, \text{lopri}, \text{scalefactor} \times \text{get_rate}(\text{cpoint}, \text{lopri})$
**Example: LatencySLO**

**H**  High Priority Workflows  \hspace{2cm}  **L**  Low Priority Workflows

**Select the high priority workflow \textit{W} with worst performance**

1. \texttt{foreach candidate in H}  
2. \texttt{miss[candidate] = latency(candidate) / guarantee[candidate]}  
3. \texttt{W = candidate in H with max miss[candidate]} 

**Weight low priority workflows by their interference with \textit{W}**

4. \texttt{foreach rsrcc in resources() // calculate importance of each resource for hipri}\n5. \texttt{importance[rsrcc] = latency(W, rsrcc) * log(slowdown(rsrc))}  
6. \texttt{foreach lopri in L // calculate low priority workflow interference}\n7. \texttt{interference[lopri] = \sum_{rsrcc} importance[rsrcc] * load(lopri, rsrcc) / load(rsrc)}

**Throttle low priority workflows proportionally to their weight**

8. \texttt{foreach lopri in L // normalize interference}  
9. \texttt{interference[lopri] /= \sum_{k} interference[k]}  
10. \texttt{foreach lopri in L}  
11. \texttt{if miss[W] > 1 // throttle}  
12. \texttt{scalefactor = 1 - \alpha \cdot (miss[W] - 1) \cdot interference[lopri]}  
13. \texttt{else // release}  
14. \texttt{scalefactor = 1 + \beta}  
15. \texttt{foreach cpoint in controlpoints() // apply new rates}\n16. \texttt{set_rate(cpoint, lopri, scalefactor * get_rate(cpoint, lopri)}}
Other types of policy...
Other types of policy...

Policy

Bottleneck Fairness
Detect most overloaded resource
Fair-share resource between tenants using it
Other types of policy...

**Bottleneck Fairness**
- Detect most overloaded resource
- Fair-share resource between tenants using it

**Dominant Resource Fairness**
- Estimate demands and capacities from measurements
Other types of policy…

**Bottleneck Fairness**
Detect most overloaded resource
Fair-share resource between tenants using it

**Dominant Resource Fairness**
Estimate demands and capacities from measurements

Concise
Any resources can be bottleneck (policy doesn’t care)
Not system specific
Retro Evaluation
Instrumentation
Instrumentation

Retro implementation in Java
  Instrumentation Library
  Central controller implementation
Instrumentation

Retro implementation in Java
Instrumentation Library
Central controller implementation

To enable Retro
- Propagate Workflow ID within application (like X-Trace, Dapper)
- Instrument resources with wrapper classes
Instrumentation

Retro implementation in Java
Instrumentation Library
Central controller implementation

To enable Retro
- Propagate Workflow ID within application (like X-Trace, Dapper)
- Instrument resources with wrapper classes

Overheads
Instrumentation

Retro implementation in Java
 Instrumentation Library
 Central controller implementation

To enable Retro

- Propagate Workflow ID within application (like X-Trace, Dapper)
- Instrument resources with wrapper classes

Overheads
 Resource instrumentation automatic using AspectJ
Instrumentation

Retro implementation in Java
  Instrumentation Library
  Central controller implementation

To enable Retro
  ✅ Propagate Workflow ID within application (like X-Trace, Dapper)
  ⚪ Instrument resources with wrapper classes

Overheads
  Resource instrumentation automatic using AspectJ
  Overall 50-200 lines per system to modify RPCs
Instrumentation

Retro implementation in Java
  Instrumentation Library
  Central controller implementation

To enable Retro
  Propagate Workflow ID within application (like X-Trace, Dapper)
  Instrument resources with wrapper classes

Overheads
  Resource instrumentation automatic using AspectJ
  Overall 50-200 lines per system to modify RPCs
  Retro overhead: max 1-2% on throughput, latency
Experiments
Experiments
Experiments

**Workflows**
- MapReduce Jobs (HiBench)
- HBase (YCSB)
- HDFS clients
- Background Data Replication
- Background Heartbeats

**Systems**
- YARN
- MapReduce
- ZooKeeper
- HBase
- HDFS
Experiments

Workflows
- MapReduce Jobs (HiBench)
- HBase (YCSB)
- HDFS clients
- Background Data Replication
- Background Heartbeats

Resources
- CPU, Disk, Network (All systems)
- Locks, Queues (HDFS, HBase)

Systems
- YARN
- MapReduce
- ZooKeeper
- HBase
- HDFS
Experiments

**Workflows**
- MapReduce Jobs (HiBench)
- HBase (YCSB)
- HDFS clients
- Background Data Replication
- Background Heartbeats

**Resources**
- CPU, Disk, Network (All systems)
- Locks, Queues (HDFS, HBase)

**Systems**
- YARN
- MapReduce
- ZooKeeper
- HBase
- HDFS

**Policies**
- Latency SLO
- Bottleneck Fairness
- Dominant Resource Fairness
Experiments

Workflows
- MapReduce Jobs (HiBench)
- HBase (YCSB)
- HDFS clients
- Background Data Replication
- Background Heartbeats

Resources
- CPU, Disk, Network (All systems)
- Locks, Queues (HDFS, HBase)

Systems
- YARN
- MapReduce
- ZooKeeper
- HBase
- HDFS

Policies
- Latency SLO
- Bottleneck Fairness
- Dominant Resource Fairness

Policies for a mixture of systems, workflows, and resources
Results on clusters up to 200 nodes
See paper for full experiment results
Experiments

Workflows
- MapReduce Jobs (HiBench)
- HBase (YCSB)
- HDFS clients
- Background Data Replication
- Background Heartbeats

Resources
- CPU, Disk, Network (All systems)
- Locks, Queues (HDFS, HBase)

Systems
- YARN
- MapReduce
- ZooKeeper
- HBase
- HDFS

Policies
- LatencySLO
- Bottleneck Fairness
- Dominant Resource Fairness

Policies for a mixture of systems, workflows, and resources
Results on clusters up to 200 nodes
See paper for full experiment results
This talk LatencySLO policy results
Experiments

**Workflows**
- MapReduce Jobs (HiBench)
- HBase (YCSB)
- HDFS clients
- Background Data Replication
- Background Heartbeats

**Resources**
- CPU, Disk, Network (All systems)
- Locks, Queues (HDFS, HBase)

**Systems**
- YARN
- MapReduce
- ZooKeeper
- HBase
- HDFS

**Policies**
- LatencySLO
- Bottleneck
- Fairness
- Dominant Resource Fairness

Policies for a mixture of systems, workflows, and resources
Results on clusters up to 200 nodes
See paper for full experiment results

**This talk** LatencySLO policy results
HDFS read 8k

HBase read 1 row

HBase read 1 cached row
**SLO-Normalized Latency vs Time [minutes]**

- **HBase Table Scan**
- **HDFS mkdir**
- **HBase Cached Table Scan**
- **HDFS read 8k**
- **HBase read 1 row**
- **HBase read 1 cached row**

**SLO Policy Enabled**

- **SLO target**
Conclusion
Conclusion

Resource management for *shared* distributed systems
Resource management for *shared* distributed systems

*Centralized* resource management
Resource management for *shared* distributed systems

**Centralized resource management**

*Comprehensive*: resources, processes, tenants, background tasks
Resource management for *shared* distributed systems

**Centralized resource management**

*Comprehensive*: resources, processes, tenants, background tasks

Abstractions for writing *concise*, *general-purpose* policies:
- Workflows
- Resources (slowdown, load)
- Control points
Resource management for *shared* distributed systems

**Centralized resource management**

**Comprehensive**: resources, processes, tenants, background tasks

Abstractions for writing *concise, general-purpose* policies:
- Workflows
- Resources (slowdown, load)
- Control points

[http://cs.brown.edu/~jcmace](http://cs.brown.edu/~jcmace)