Transient Cloud Computing

Prateek Sharma
Cloud VM Pricing

- Conventional **on-demand** instances: fixed per-hour/second pricing
- **Reserved** instances: Long-term lease (1/3 years), cheaper than on-demand
- **Transient** instances: Price and availability varies over time

- Classic example: EC2 spot instances
- Price set by continuous second-price auction
- If price > user’s bid, the instance is terminated after 2 minutes
**New Paradigm: Transient Computing**

**Conventional resources**
- Continuous availability

**Transient Resource Availability**
- Access unilaterally revoked
- Applications face disruption
  - Cannot assume continuous availability
Diversity In Transient Cloud Servers

- Transient server revocations depend on server-type, location
  - >5,000 EC2 spot markets

Ex: \{server-type: m4.small, region: us-east-1, data-center-zone: A, OS: Linux\}

In EC2, price volatility $\rightarrow$ revocations

Transient cloud servers: diversity in demands and revocation frequency
Transiency Is Common In The Cloud

All major cloud providers offer transient servers

<table>
<thead>
<tr>
<th></th>
<th>Amazon EC2 Spot Instances</th>
<th>Google Cloud Preemptible VMs</th>
<th>Azure Batch VMs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lifetime</td>
<td>2-48 hours</td>
<td>&lt;24 hours</td>
<td>~12 hours</td>
</tr>
<tr>
<td>Discount</td>
<td>50-90%</td>
<td>70%</td>
<td>80%</td>
</tr>
</tbody>
</table>

- Revocations ➔ Servers have limited lifetimes
- Conventional cloud servers: non-revocable on-demand servers

![Diagram](image)
Transient Server Characteristics

- **Frequent revocations**: MTTFs of hours/days, not years
  - Run applications without disruption/performance degradation?

- **Advance warning**: Not sudden fail-stop failures
  - How to mask revocations to reduce downtimes?

- **Heterogeneity**: Different price vs. availability tradeoffs
  - Resource management policies to manage revocation risk?

- How can applications make effective use of transient resources?
- Can we design systems with transiency-specific mechanisms and policies?
Transiency-driven System Design Challenges

1. Reduce impact of revocations on availability and performance
2. Reduce number and frequency of revocations
3. Abstractions for transient servers
4. Transient resource reclamation to avoid revocations
Migrating Application State

Basic idea:
Run on spot when possible. Migrate to on-demand when revoked.

- Existing technique: VM Live Migration
  - Migration may not complete within advance warning (2 mins)
  - Incomplete migrations result in state-loss and unavailability

- Can we completely migrate VMs within warning period?
SpotCheck VM Migration

- **Bounded-time VM Live Migration**
  - Ensures VMs migrate within a specified time duration
  - Independent of memory size, application behavior

- **Spot**
  - Continuously checkpoint memory
  - Residual dirty pages sent in bounded time

- **Backup Server**
  - Restore skeleton state immediately
    - VCPU state, page-tables,..
    - Copy remaining pages on access

- **On-demand**
SpotCheck: A Derivative Cloud

- Derivative cloud: Cloud middleware derived from native cloud
- SpotCheck: Illusion of low-cost, non-revocable servers to run unmodified apps
- Multiplex spot and on-demand pools across multiple customers
Implementing SpotCheck

- How to migrate VM state in public clouds?
  - Migration and other hypervisor functionality not exposed
  - Solution: Nested Virtualization (Xen-Blanket)
  - Bonus: can run multiple nested VMs

- Mitigating concurrent revocations:
  - Map customer VMs to different spot servers
  - Map VMs from different spot servers to a backup server
SpotCheck Application Performance

- Performance and cost overhead of continuous checkpointing is low

<table>
<thead>
<tr>
<th>SpecJBB</th>
<th>0.015%</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPC-W</td>
<td>16.7%</td>
</tr>
</tbody>
</table>

- Performance degradation due to continuous checkpointing is low

- Backup servers can support ~40 VMs
- Amortizes backup server cost
Challenges in Running Distributed Applications

- All servers concurrently revoked → complete resource starvation
- Can we mitigate concurrent revocations for distributed apps?
Key Idea: Select Heterogenous Servers

- Heterogenous servers: different configuration, data center racks/zones
- Many applications can tolerate partial failures, run in degraded mode
- Key: Uncorrelated revocations
  - Can we use heterogenous server selection in the cloud?
Heterogenous Server Selection

- Select a heterogenous collection of servers that:
  - Minimizes cost
  - Minimizes number and frequency of revocations
  - Minimizes fraction lost due to revocation ➔ Uncorrelated servers
Server Portfolios: Heterogenous Mix of Servers

Server selection is analogous to financial portfolio construction

- Stocks, bonds to max returns and min risk
- Reduce risk of large losses using uncorrelated assets
- Investors have different risk and reward preferences

---

Server types to max savings and min revocation risk

Reduce risk of concurrent revocation using uncorrelated servers

Applications have different risk/reward preferences
Key Idea: Diversification

- Diversification reduces volatility due to individual markets

Many EC2 markets have low correlations

Diversification is a viable strategy
Model-driven Portfolio Construction

• Based on Modern Portfolio Theory from finance (Markowitz 1953)

• Objective: Maximize risk-adjusted returns

\[ E[\text{Return}] - \alpha \cdot \text{Risk} \]

1 - \( \frac{E[\text{Spot-price}]}{\text{On-demand-price}} \)

Risk averseness parameter \([0, \infty)\)

Revocation risk (variance in prices)

• Example output: portfolio allocation vector, \(x\):

<table>
<thead>
<tr>
<th></th>
<th>small</th>
<th>med</th>
<th>large</th>
</tr>
</thead>
<tbody>
<tr>
<td>Example</td>
<td>0</td>
<td>0.2</td>
<td>0.8</td>
</tr>
</tbody>
</table>


Portfolio Construction Optimization

\[ E[\text{Return}] - \alpha \cdot \text{Risk} \]

\[ 1 - \frac{E[\text{Spot-price}]}{\text{On-demand-price}} \]

Risk averseness parameter 

\([0, \text{inf})\]

Revocation risk 
(variance in prices)

\( c: \text{Discount vector} \)

\( \times \)

\( \text{Spot price traces} \)

Maximize: \( cx^T - \alpha xVx^T \)

Subject to: \( \sum_{i=1}^{n} x_i = 1 \)

\( x \geq 0 \)

Convex Quadratic

\( \frac{1}{n} \sum_{t=1}^{n} ((A(t) - E[A])(B(t) - E[B])) \)

\( \text{Spot price traces for markets A,B} \)
Risk-Return Tradeoffs With Portfolios

\[ E[\text{Return}] - \alpha \cdot \text{Risk} \]

- Failures/delays are undesirable
- High diversification
  - Lower savings, low risk
- Failures/delays are tolerable
- Low diversification
  - High savings, higher risk
ExoSphere: Transiency-aware Cluster Management

- ExoSphere provides virtual clusters to run multiple applications
  - Spark, MPI, BOINC
- Transiency-aware Mesos (4K lines)
- Applications submit: (#CPUs, Mem, \( \alpha \))
- Applications can share cloud servers
  - Multiplexing \( \rightarrow \) Reduced costs
- Applications get price, MTTF, revocation-warning notifications
Transiency Mitigation Policies With ExoSphere

- ExoSphere enables applications to implement custom policies
- Especially useful for fault-tolerance
  - Checkpoint application state and roll-back in case of revocations
  - Use existing mechanisms to implement policy in few lines of code

Young-Daly periodic checkpoint interval = \( \sqrt{2 \times \text{Time to Checkpoint} \times \text{MTTF}} \)

Checkpoint size, disk-speed Provided by ExoSphere
When To Checkpoint?

- **Key idea**: checkpoint periodically to minimize expected running time
- Simplified Spark performance model on transient servers

\[ E[T] = T + \frac{T}{\tau} \cdot \delta + \frac{T}{MTTR} \left(\frac{\tau}{2}\right) \]

- **Checkpointing overhead**
- **Recovery**

\[ \tau: \text{Checkpointing interval} \]
\[ \delta: \text{Time required to save RDD} \]
\[ MTTR: \text{Mean Time To Revocation} \]

- **Minimize** \( E[T] \) **with respect to** \( \tau \):

\[ \text{RDD checkpoint interval} \quad (\tau) = \sqrt{2 \cdot \delta \cdot MTTR} \]

- RDD size, write speed
- Spot market price traces

Young-Daly 1974
Effectiveness of Portfolios

Cost-risk comparison based on EC2 spot prices from March-Nov 2015

- 85% cost savings compared to on-demand
- ~100x reduction in revocation risk compared to existing approaches
Application Performance In ExoSphere

- Portfolios+checkpointing: reduces transiency overhead to < 10%
- Risk intolerant interactive applications see significant performance benefit
ExoSphere Summary

- Transient server selection based on portfolio modeling
- ExoSphere: system for portfolio based cluster management
- General framework supporting common transiency policies
- Cloud transient servers are an increasingly popular area:

<table>
<thead>
<tr>
<th>Homogenous server selection</th>
<th>OptiSpot, SpotOn [SoCC ’15],…</th>
</tr>
</thead>
<tbody>
<tr>
<td>Heterogenous server selection</td>
<td>Amazon SpotFleets, Tributary [ATC’18]</td>
</tr>
<tr>
<td>Application-specific techniques</td>
<td>Spark [TR-Spark-SoCC ’16, HPDC’17], MPI [HPDC ’14], ML [Proteus-EuroSys’17]</td>
</tr>
</tbody>
</table>
Reclaiming Resources Using Resource Deflation

- How to reclaim resources from low-priority VMs?
  - **Resource Deflation**: Fractional resource reclamation.

- For most applications: higher availability > performance degradation
- Trade-off higher availability for performance degradation

---

VMs shrunk: -50% -50%

Cost

Deflatable

Spot

Availability

On-demand

Performance
Questions?

- Transient Servers
- SpotCheck
- Flint
- ExoSphere
- Resource Deflation
ExoSphere Backup

Cost Saving (%)

Portfolio
Spot Fleet
Greedy

Revocation Risk Probability

Market weight

Revocation risk averseness (alpha)
ExoSphere + Flint

**Graph 1:**
- X-axis: Number of Failures
- Y-axis: Running Time (s)
- Legend: Checkpointing, Recomputation

**Graph 2:**
- Two bars for each distribution:
  - Private
  - Shared
- Distributions: All-low, Equal-distr, 1:2:1, All-high
- Y-axis: Total Cost ($)

**Graph 3:**
- Checkpointing intervals: No-Ckpting, 720, 480, 360, All-RDDs, System-level
- Y-axis: Running Time (s)
- X-axis: Checkpointing Interval (s)
• RDD checkpointing: <10% performance overhead
• System-level checkpointing: high overhead of writing OS and cache
• Greedy selection only considers cost → high revocation risk
• Risk intolerant applications see significant performance benefit
Interactive Data Processing Performance

Default Spark

- No revocations (On-demand server performance)
- With server revocation

Flint

- Interactivity maintained even after revocation

TPC-H response time (s)

10x reduction