SOL: Safe On-Node Learning in Cloud Platforms

Yawen Wang\textsuperscript{1}, Dan Crankshaw\textsuperscript{2}, Neeraja Yadwadkar\textsuperscript{3}, Daniel Berger\textsuperscript{2}, Christos Kozyrakis\textsuperscript{1}, Ricardo Bianchini\textsuperscript{2}

\textsuperscript{1}Stanford University \textsuperscript{2}Microsoft Research \textsuperscript{3}University of Texas at Austin

\textbf{Platform Lab Retreat}

\textbf{2/17/2022}
Talk Outline

- **Advantages** of on-node ML in the cloud
- **Study of 77 Azure node agents** and how on-node ML can benefit them
- **Challenges** of deploying on-node ML in production
- **Proposed solution:** SOL - a **Safe On-node Learning framework**
  - Insights behind the SOL design
  - How to develop agents using SOL API & Runtime
- **Evaluation** of smart node management agents built with SOL
On-Node Machine Learning in the Cloud

- **Advantages:**
  - **Fresh** workload-tuned models and predictions
    - Patterns of workloads change frequently
    - Off-node training/inference not fast enough
  - **Fine-grained** telemetry
    - Captures high frequency workload dynamics
    - Expensive to ship off-node

- **Recent work for on-node ML:**

<table>
<thead>
<tr>
<th>Agent</th>
<th>Goal</th>
<th>Action</th>
<th>Frequency</th>
<th>Inputs</th>
<th>Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>SmartHarvest [1]</td>
<td>Harvest idle cores</td>
<td>Core assignment</td>
<td>25 ms</td>
<td>CPU usage</td>
<td>Cost-sensitive classification</td>
</tr>
<tr>
<td>Hipster [2]</td>
<td>Reduce power draw</td>
<td>Core assignment &amp; frequency</td>
<td>1 s</td>
<td>App QoS and load</td>
<td>Reinforcement learning</td>
</tr>
<tr>
<td>LinnOS [3]</td>
<td>Improve IO perf</td>
<td>IO request routing/rejection</td>
<td>Every IO</td>
<td>Latencies, queue sizes</td>
<td>Binary classification</td>
</tr>
<tr>
<td>ESP [4]</td>
<td>Reduce interference</td>
<td>App scheduling</td>
<td>Every app</td>
<td>App run time, perf counters</td>
<td>Regularized regression</td>
</tr>
</tbody>
</table>

[1] Wang et al. “SmartHarvest: harvesting idle CPUs safely and efficiently in the cloud” (EuroSys’21)
Azure Node Agents in Production

• A total of 77 node agents
  • run on the order of seconds to months

• 7 categories
  • Configuration, services, monitoring/logging, watchdogs, resource control, and operator access

• Agents that collect data to make decisions based on current environment and workload can leverage ML

<table>
<thead>
<tr>
<th>Category</th>
<th>Benefits of ML</th>
</tr>
</thead>
<tbody>
<tr>
<td>Monitoring</td>
<td>Dynamically select optimal sampling frequency to minimize cost</td>
</tr>
<tr>
<td>Watchdogs</td>
<td>Learn from node data to detect problems and take actions</td>
</tr>
<tr>
<td>Resource control</td>
<td>Learn workload dynamics to improve node utilization/performance</td>
</tr>
</tbody>
</table>
Example of On-Node Learning

• **SmartOverclock agent**
  • Leverages Q-learning to dynamically overclock CPU
  • Goal is to maximize workload performance while not wasting power
  • Collects telemetry and changes CPU frequency every 1 second

Common workflow for ML-based node agent

- Collect Data: Reads Instructions Per Second (IPS)
- Update Model: Computes reward and updates Q table
- Model Predict: Selects CPU frequency with Q-Learning
- Take Action: Sets VM cores to selected frequency
Challenges of On-Node Learning
Challenges of On-Node Learning

<table>
<thead>
<tr>
<th>Challenge</th>
<th>Example in SmartOverclock</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bad input data</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Reduction in Workload Completion Time with SmartOverclock

- All valid data: 41%
- 25% invalid data: 0%
- 75% invalid data: 6%
## Challenges of On-Node Learning

<table>
<thead>
<tr>
<th>Challenge</th>
<th>Example in SmartOverclock</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bad input data</td>
<td>Corrupted counter readings</td>
<td>Explicitly check data assumptions</td>
</tr>
</tbody>
</table>

Poor model accuracy
Challenges of On-Node Learning

<table>
<thead>
<tr>
<th>Challenge</th>
<th>Example in SmartOverclock</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bad input data</td>
<td>Corrupted counter readings</td>
<td>Explicitly check data assumptions</td>
</tr>
<tr>
<td>Poor model accuracy</td>
<td>Mis-calibrated model</td>
<td>Continuously evaluate model accuracy</td>
</tr>
<tr>
<td>Unpredictable compute availability</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
# Challenges of On-Node Learning

<table>
<thead>
<tr>
<th>Challenge</th>
<th>Example in SmartOverclock</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bad input data</td>
<td>Corrupted counter readings</td>
<td>Explicitly check data assumptions</td>
</tr>
<tr>
<td>Poor model accuracy</td>
<td>Mis-calibrated model</td>
<td>Continuously evaluate model accuracy</td>
</tr>
<tr>
<td>Unpredictable compute availability</td>
<td>Stale overclocking prediction</td>
<td>Account for stale data/predictions</td>
</tr>
<tr>
<td>Maintaining node performance and reliability</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Challenges of On-Node Learning

<table>
<thead>
<tr>
<th>Challenge</th>
<th>Example in SmartOverclock</th>
<th>Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bad input data</td>
<td>Corrupted counter readings</td>
<td>Explicitly check data assumptions</td>
</tr>
<tr>
<td>Poor model accuracy</td>
<td>Mis-calibrated model</td>
<td>Continuously evaluate model accuracy</td>
</tr>
<tr>
<td>Unpredictable compute availability</td>
<td>Stale overclocking prediction</td>
<td>Account for stale data/predictions</td>
</tr>
<tr>
<td>Maintaining node performance and reliability</td>
<td>Wasting power</td>
<td>Restore node to a safe state when problems are detected</td>
</tr>
</tbody>
</table>

**SOL:** a **Safe On-node Learning** framework that helps development of **safe ML-based agents** robust under a variety of **failure conditions** from production.
**Insights for SOL**

- **Node agents** are **heterogenous**
  - Read different counters
  - Perform different tasks
  - Operate at different timescale

- **Agents structure and failure conditions** are **shared**
  - Operations for ML-based control agents
    - Collect data
    - Update model
    - Model predict
    - Take action
  - Types of problem to detect and mitigate in production
    - Bad input data
    - Poor model accuracy
    - Unpredictable compute availability
    - Impact on node performance and reliability

**Agent-specific logic**

**SOL API & runtime**
SOL Design Overview
SOL Design #1: Common Agent Task Abstraction

Collect Data → Update Model → Model Predict → Take Action

🌟 Identify common operations across on-node learning agents
SOL Design #2: Decoupling of ML and Actuation Logic

Split into two components: Model and Actuator
SOL Design #2: Decoupling of ML and Actuation Logic

- **Model** and **Actuator** run in separate threads
- **Actuator** is not blocked by delays from **Model**

Take a default action if prediction is unavailable

Prediction Queue

Collect Data → Update Model → Model Predict → Take Action
SOL Design #3: Integrating Watchdog-Style Safeguards

Add safeguards to automatically detect and mitigate problems in production
SOL Design #3: Integrating Watchdog-Style Safeguards

- Collect Data
- Validate Data
- Update Model
- Model Predict

Prediction Queue

Take Action

Pass

Fail

Default Prediction
SOL Design #3: Integrating Watchdog-Style Safeguards
Key elements of SOL API
1. ML-based control operations
2. Watchdog-style safeguards

Implementation provided by agent developers

SOL API

CollectData() -> ValidateData() -> UpdateModel() -> AssessModel()

ModelPredict() -> DefaultPredict()

Prediction Queue

AssessPerformance() -> TakeAction() -> Mitigate()

Pass -> Fail

Pass -> Fail

Implementation provided by agent developers
Running a SOL Agent

```c++
void main()
{
    Schedule schedule(config_file);
    Model* model = new OverclockModel();
    Actuator* act = new OverclockActuator();
    SOL::RunAgent(model, act, schedule);
}
```

Developer-provided configuration parameters
Experimental Setup

• Experiments are conducted on **single server node**
• **SOL agents** run in user-space of Hyper-V
• **SOL runtime** runs in the same process as the agent
• Different **failures** are injected to evaluate SOL agents robustness
Evaluation

• We used SOL to implement three on-node resource management agents

<table>
<thead>
<tr>
<th>Agent</th>
<th>Goal</th>
<th>Action</th>
<th>Frequency</th>
<th>Inputs</th>
<th>Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>SmartOverclock</td>
<td>Improve VM performance</td>
<td>CPU overclocking for VMs</td>
<td>1 s</td>
<td>IPS, CPU usage</td>
<td>Reinforcement Learning</td>
</tr>
<tr>
<td>SmartHarvest</td>
<td>Harvest idle cores</td>
<td>CPU core assignment among primary VMs and</td>
<td>25 ms</td>
<td>CPU usage data</td>
<td>Cost-sensitive multi-class classification</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ElasticVM</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SmartMemory</td>
<td>Reduce memory page scanning cost</td>
<td>Cold memory page for migration</td>
<td>9.6 s</td>
<td>Memory access bit scans</td>
<td>Multi-armed bandits</td>
</tr>
</tbody>
</table>
Evaluation

- We used SOL to implement three on-node resource management agents

<table>
<thead>
<tr>
<th>Agent</th>
<th>Goal</th>
<th>Action</th>
<th>Frequency</th>
<th>Inputs</th>
<th>Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>SmartOverclock</td>
<td>Improve VM performance</td>
<td>CPU overclocking for VMs</td>
<td>1 s</td>
<td>IPS, CPU usage</td>
<td>Reinforcement Learning</td>
</tr>
<tr>
<td>SmartHarvest</td>
<td>Harvest idle cores</td>
<td>CPU core assignment among primary VMs and ElasticVM</td>
<td>25 ms</td>
<td>CPU usage data</td>
<td>Cost-sensitive multi-class classification</td>
</tr>
<tr>
<td>SmartMemory</td>
<td>Reduce memory page scanning cost</td>
<td>Cold memory page for migration</td>
<td>9.6 s</td>
<td>Memory access bit scans</td>
<td>Multi-armed bandits</td>
</tr>
</tbody>
</table>
Evaluation of SmartOverclock

1. How does SmartOverclock compare with static policies?
2. How well does SOL mitigate impact from invalid data?
3. How well does SOL mitigate impact from delayed prediction?
4. How well does SOL mitigate impact of inaccurate model?
5. How well does SOL prevent wasting power when the server is idle?
Comparison with Static Policies

<table>
<thead>
<tr>
<th>1.5GHz (nominal frequency)</th>
<th>2.3GHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.9GHz</td>
<td>SmartOverclock</td>
</tr>
</tbody>
</table>
Comparison with Static Policies

Higher is better

Lower is better

Normalized Performance

1.5GHz (nominal frequency)

2.3GHz

1.9GHz

SmartOverclock

Normalized Power

Synthetic ObjectStore DiskSpeed

Synthetic ObjectStore DiskSpeed
Comparison with Static Policies

Higher is better

Lower is better

Normalized Performance

1.5GHz (nominal frequency)  2.3GHz  1.9GHz

Normalized Power

1.00  2.03  1.61

Synthetic  ObjectStore  DiskSpeed

SmartOverclock

(28)
Comparison with Static Policies

- Higher is better
- Lower is better

### 1.5GHz (nominal frequency)
- Synthetic: 1.00
- ObjectStore: 1.00
- DiskSpeed: 1.00

### 2.3GHz
- Synthetic: 1.00
- ObjectStore: 1.00
- DiskSpeed: 1.00

### SmartOverclock
- Synthetic: 1.00
- ObjectStore: 1.00
- DiskSpeed: 1.00
Comparison with Static Policies

Higher is better

Lower is better

Normalized Performance

Normalized Power
Comparison with Static Policies

SmartOverclock agent achieves near optimal performance and power usage on a variety of workloads.
Invalid Data

Return random synthetic out-of-range IPS data to the agent
Invalid Data

Without data validation safeguard, 5% of invalid IPS readings → 17% drop in performance

Return random synthetic out-of-range IPS data to the agent

Synthetic Workload

Lower is Better

Ideal Setting

Higher is Better
Invalid Data

Synthetic Workload

- 5% Invalid
- 10% Invalid
- 25% Invalid

Return random synthetic out-of-range IPS data to the agent

Graph:
- X-axis: Normalized Performance
- Y-axis: Normalized Power
- Ideal Setting
- More invalid data: Worse performance
- Lower is Better
- Higher is Better
Invalid Data

Synthetic Workload

- 5% Invalid
- 5% Invalid (safeguard)

Data-validation safeguard drops out-of-range IPS data to preserves model accuracy.

Return random synthetic out-of-range IPS data to the agent.

Lower is Better

Higher is Better

Ideal Setting

Better performance w/ data validation.
Invalid Data

Return random synthetic out-of-range IPS data to the agent

Data-validation safeguard drops out-of-range data to preserve model accuracy

Better performance w/ data validation

Lower is Better

Higher is Better
Invalid Data

Synthetic Workload

- 25% Invalid
- 25% Invalid (safeguard)

Invalid Data

Data-validation safeguard drops out-of-range IPS data to preserves model accuracy

Ideal Setting

Lower is Better

Higher is Better

Impact on agent performance

Return random synthetic out-of-range IPS data to the agent
Delayed Prediction

Inject delay in the model thread during workload phase change
Delayed Prediction

Synthetic Workload

+ 36% power

Blocking Actuator

Inject delay in the model thread during workload phase change

The blocking actuator waits until a prediction is available from the model, wastes power when prediction is delayed
Delayed Prediction

Synthetic Workload

Inject delay in the model thread during workload phase change

The non-blocking actuator waits for up to 5 sec before restoring the node to a safe state (nominal frequency)
Inaccurate Model

A broken model that always overclocks to the highest frequency
Inaccurate Model

A broken model that always overclocks to the highest frequency

Inaccurate Model

- Synthetic
- ObjectStore
- DiskSpeed

Normalized Power

Normalized Performance

Wasting power
Inaccurate Model

A broken model that always overclocks to the highest frequency

The model safeguard detects overclocking without gains and avoids wasting power using the default action.
Idle Server

Long-term low CPU utilization phase during the workload run
Idle Server

Long-term low CPU utilization phase during the workload run

ObjectStore

Without Actuator Safeguard

long idle phase

Time (seconds)
Idle Server

ObjectStore

The actuator safeguard detects long idle periods and completely disables overclocking to avoid wasting power.
Conclusion

• We present **SOL** for developing **safe on-node learning agents**
  • A general and extensible framework
  • Facilitates deployment and operation of safe agents under a wide variety of realistic issues in production
    • Bad data readings
    • Inaccurate model
    • Scheduling delays
    • Node performance and reliability

• We implement three agents in SOL and show
  • The benefits of incorporating ML into on-node resource management tasks
  • The design of SOL ensures agents are robust to different failure conditions

Thank you!

**Contact:** Yawen Wang  
✉️ yawenw@stanford.edu