RRC: Responsive Replicated Containers

Diyu Zhou*
UCLA and EPFL

Yuval Tamir
UCLA

*Looking for a faculty job
Server Applications Need Responsive Fault Tolerance

Server Applications:

• Low latency
• High throughput
• High reliability
Server Applications Need Responsive Fault Tolerance

Server Applications:

• Low latency
• High throughput
  → Multithreading
High reliability
  → Fault Tolerance
Server Applications Need Responsive Fault Tolerance

Server Applications:

• Low latency
• High throughput
  → Multithreading

High reliability
  → Fault Tolerance

Fault Tolerance Mechanism Requirements

• Low latency overhead
• Maintain high throughput
  – Low throughput overhead
  – Support multithreading
Server Applications Need Responsive Fault Tolerance

Server Applications:
- Low latency
- High throughput → Multithreading
- High reliability → Fault Tolerance

Fault Tolerance Mechanism Requirements
- Low latency overhead
- Maintain high throughput
  - Low throughput overhead
  - Support multithreading
- Minimize development cost
  - No code modification
  - Compatibility with existing clients → Application Transparency
Replication → Application-Transparent Fault Tolerance
Replication → Application-Transparent Fault Tolerance

Client

Primary host

Server App
Replication Runtime

Sync

Backup host

Server App
Replication Runtime
Replication → Application-Transparent Fault Tolerance

Client

Primary host

Server App
Replication Runtime

Failure

Sync

Backup host

Server App
Replication Runtime
Replication → Application-Transparent Fault Tolerance

Client

Primary host

Server App Replication Runtime

Failure

Backup host

Server App Replication Runtime
Replication is Old News

Bell Systems No. 1 ESS (1964)

Stratus/32 multiprocessor node (1983)

IBM G5/G6 Processing Unit (1999)

Remus: Virtual Machine Replication (2008)
What is Missing in Existing Replication Schemes?

• Many older schemes:
  – Require customized hardware
  – No support for multithreaded applications
What is Missing in Existing Replication Schemes?

• Many older schemes:
  – Require customized hardware
  – No support for multithreaded applications

• Schemes based on checkpointing to a **passive** backup
  – Unacceptable high latency overhead
What is Missing in Existing Replication Schemes?

• Many older schemes:
  – Require customized hardware
  – No support for multithreaded applications

• Schemes based on checkpointing to a passive backup
  – Unacceptable high latency overhead

• Schemes based on active replication
  – Untracked nondeterministic events (e.g., data races)
    Unpredictable slowdown during normal operation (with some schemes)
    Recovery failure (with some schemes)
  – Performance limited by tight coupling among replicas.
What is Missing in Existing Replication Schemes?

• Many older schemes:
  – Require customized hardware
  – No support for multithreaded applications

• Schemes based on checkpointing to a **passive** backup
  – Unacceptable high latency overhead

• Schemes based on **active** replication
  – Untracked nondeterministic events (e.g., data races)
    Unpredictable slowdown during normal operation (with some schemes)
    Recovery failure (with some schemes)
  – Performance limited by tight coupling among replicas.

RRC overcomes limitations by **decoupling** replication-related operations from normal operations
Talk Outline

• Preface
• **Motivation**
• RRC overview
• Overcoming design and implementation challenges
• Evaluation
Passive Backup: Checkpointing-Based Mechanisms

Primary

Application

Backup Host

Client

Epoch 0

Execute
Pause
Checkpoint
Send state

Epoch 1

Execute
Wait for ACK
Release output
Pause
Passive Backup: Checkpointing-Based Mechanisms

Primary

Application

Epoch 0

Execute

Pause

Checkpoint

Send state

Epoch 1

Execute

Wait for ACK

Release output

Pause

Backup Host

Client
Passive Backup: Checkpointing-Based Mechanisms

Application

Buffer

Primary

External Output

Epoch 0

Execute

Pause

Checkpoint

Send state

Epoch 1

Execute

Wait for ACK

Release output

Pause

Application

Backup Host

Client
Passive Backup: Checkpointing-Based Mechanisms

Application

External Output

Buffer

Primary

Epoch 0

Execute

Pause

Check point

Send state

Epoch 1

Execute

Wait for ACK

Release output

Pause

Backup Host

Client
Passive Backup: Checkpointing-Based Mechanisms

1. **External Output**

2. **Checkpoint**

**Application**
- Execute
- Pause
- Checkpoint

**Buffer**
- Send state

**Backup Host**

**Client**

**Epoch 0**
- Execute
- Pause

**Epoch 1**
- Execute
- Pause
- Wait for ACK
- Release output
Passive Backup: Checkpointing-Based Mechanisms

1. Application
2. Checkpoint
3. Ack

Primary

Application

Epoch 0

- Execute
- Pause
- Checkpoint
- Send state

Epoch 1

- Execute
- Pause
- Wait for ACK
- Release output
Passive Backup: Checkpointing-Based Mechanisms

1. External Output
2. Checkpoint
3. Ack
4. Release

Primary

Application

Buffer

Backup Host

Client

Epoch 0

Execute

Pause

Checkpoint

Send state

Wait for ACK

Pause

Epoch 1

Execute

Release output
Passive Backup: Checkpointing-Based Mechanisms

Why delayed output:
Backup needs to restore state consistent with clients
Checkpointing-Based Mechanisms → High latency Overhead

- Output delay = remaining execute time in Epoch 0 + time up to receipt of ACK in Epoch 1
### Checkpointing-Based Mechanisms → High latency Overhead

<table>
<thead>
<tr>
<th>Application</th>
<th>Epoch 0</th>
<th></th>
<th>Epoch 1</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Execute</td>
<td>Pause</td>
<td>Execute</td>
<td>Pause</td>
<td></td>
</tr>
<tr>
<td>Checkpoint</td>
<td>Send state</td>
<td><em>Wait for ACK</em></td>
<td>Release output</td>
<td></td>
</tr>
</tbody>
</table>

- Output delay = remaining execute time in Epoch 0 + time up to receipt of ACK in Epoch 1
- Checkpointing is expensive → Critical checkpointing (epoch) interval tradeoff
  - *Short* interval → *High* throughput overhead, *low* latency overhead
  - *Long* interval → *Low* throughput overhead, *high* latency overhead
Checkpointing-Based Mechanisms $\rightarrow$ High latency Overhead

- Output delay = remaining execute time in Epoch 0 + time up to receipt of ACK in Epoch 1
- Checkpointing is expensive $\rightarrow$ Critical checkpointing (epoch) interval tradeoff
  - *Short* interval $\rightarrow$ *High* throughput overhead, *low* latency overhead
  - *Long* interval $\rightarrow$ *Low* throughput overhead, *high* latency overhead

In practice: 10s of milliseconds interval $\rightarrow$ 10s of milliseconds latency

$\rightarrow$ Unacceptably high latency overhead
Active Backup: Mechanisms based on Active Replication

- Primary and backup execute application code
Active Backup: Mechanisms based on Active Replication

- Primary and backup execute application code
- Primary sends outcomes of nondeterministic events to backup
Active Backup: Mechanisms based on Active Replication

- Primary and backup execute application code
- Primary sends outcomes of nondeterministic events to backup
- Backup enforces outcome of nondeterministic events to match execution
Disadvantages of Active Backup Mechanisms

Backup execution must be consistent with primary:

1. Nondeterministic event log
2. Replay
Disadvantages of Active Backup Mechanisms

Backup execution must be consistent with primary:

➔ Consequences of untracked nondeterministic events (e.g., data races):
  ⦿ Unpredictable slowdowns during normal operation (for some mechanisms)
  ⦿ Recovery failure (for some mechanisms)
Disadvantages of Active Backup Mechanisms

Backup execution must be consistent with primary:

→ Consequences of untracked nondeterministic events (e.g., data races):
  ←Unpredictable slowdowns during normal operation (for some mechanisms)
  ←Recovery failure (for some mechanisms)

• Performance limited by tight coupling between replicas

• Resource overhead lower bound = 100%
Undesirable Couplings in Current Mechanisms

Root cause: couplings between replication-based ops and *normal ops*
Undesirable Couplings in Current Mechanisms

Root cause: couplings between replication-based ops and *normal ops*

- Passive backup mechanisms:
  - Checkpoint interval ↔ delay in releasing outputs
  - Time to take a checkpoint ↔ service interruption
Undesirable Couplings in Current Mechanisms

Root cause: couplings between replication-based ops and *normal ops*

- Passive backup mechanisms:
  - Checkpoint interval ↔ delay in releasing outputs
  - Time to take a checkpoint ↔ service interruption

- Active backup mechanisms:
  - Untracked nondeterminism ↔ service interruption
  - Performance on the primary ↔ performance on the backup
Undesirable Couplings in Current Mechanisms

Root cause: couplings between replication-based ops and *normal ops*

• Passive backup mechanisms:
  – Checkpoint interval ↔ delay in releasing outputs
  – Time to take a checkpoint ↔ service interruption

• Active backup mechanisms:
  – Untracked nondeterminism ↔ service interruption
  – Performance on the primary ↔ performance on the backup

RRC breaks these couplings
Talk Outline

• Preface
• Motivation
• RRC overview
• Overcoming design and implementation challenges
• Evaluation
Passive Backup as the Starting Point

Passive Backup
- Primary
- Backup

Active Backup
- Primary
- Backup
Passive Backup as the Starting Point

- Avoid vulnerability to nondeterminism
- Avoid coupling performance of primary with backup
- Reduce resource overhead
Decoupling Latency Overhead from Checkpoint Interval
Using hybrid replication

Passive backup mechanisms: High latency overhead (10s of milliseconds)

Root cause: Coupling of latency overhead and checkpointing interval
Decoupling Latency Overhead from Checkpoint Interval
Using hybrid replication

Passive backup mechanisms: High latency overhead (10s of milliseconds)

Root cause: Coupling of latency overhead and checkpointing interval

Solution: Hybrid replication – combine checkpointing with execution replay
  • Outputs release decoupled from checkpoint commitment
Decoupling Latency Overhead from Checkpoint Interval
Using hybrid replication

Passive backup mechanisms: High latency overhead (10s of milliseconds)

Root cause: Coupling of latency overhead and checkpointing interval

Solution: Hybrid replication – combine checkpointing with execution replay
- Outputs release decoupled from checkpoint commitment
- On primary failure
  - Restore the last checkpoint on backup
  - Backup replays primary execution up to the last released outputs
Choice of Granularity of Replication

Virtual machine
- Process
- OS
- VMM

Process
- Process
- OS

Container
- Process
- Namespace
- OS
Choice of Granularity of Replication

- Virtual machine
  - Process
  - OS
  - VMM

- Process
  - Process
  - OS

- Container
  - Process
  - Namespace
  - OS

High runtime overheads

Tracking OS

nondeterministic events
Choice of Granularity of Replication

Virtual machine
- Process
- OS
- VMM

High runtime overheads
Tracking OS
nondeterministic events

Process
- Process
- OS

Naming conflicts
e.g., process ID

Container
- Process
- Namespace
- OS
Choice of Granularity of Replication

Virtual machine
- Process
- OS
- VMM

Process
- Process
- OS

Container
- Process
- Namespace
- OS

1. High runtime overheads
2. Tracking OS nondeterministic events

Naming conflicts e.g., process ID

Resolves limitations of processes/ VMs
Normal operation

<table>
<thead>
<tr>
<th>Epoch 0</th>
<th>Epoch 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execute</td>
<td>Stop</td>
</tr>
<tr>
<td>Check point</td>
<td>Send state</td>
</tr>
<tr>
<td>Execute</td>
<td></td>
</tr>
</tbody>
</table>

Application

Container

Primary

Backup
Normal operation

<table>
<thead>
<tr>
<th>Epoch 0</th>
<th>Epoch 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execute</td>
<td>Execute</td>
</tr>
<tr>
<td>Stop</td>
<td></td>
</tr>
<tr>
<td>Checkpoint</td>
<td>Send state</td>
</tr>
</tbody>
</table>

Application

Primary

Backup
Normal operation

Epoch 0
- Execute
- Stop
- Check point

Epoch 1
- Execute
- Send state

Application
Container
Primary
Backup
Normal operation

Epoch 0
- Execute
- Stop
- Checkpoint

Epoch 1
- Execute
- Send state

1. Request

- Application
- Container
- Primary

- PackRec
- Backup

2. Record
Normal operation

<table>
<thead>
<tr>
<th>Epoch 0</th>
<th>Epoch 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execute</td>
<td>Execute</td>
</tr>
<tr>
<td>Stop</td>
<td>Send state</td>
</tr>
<tr>
<td>Check point</td>
<td></td>
</tr>
</tbody>
</table>

1. Request
2. Record

Application

Container

Primary

Backup

PackRec
Normal operation

Epoch 0
- Execute
- Stop
- Checkpoint

Epoch 1
- Execute
- Send state

1. Request

Primary
- Application
- Container

Backup
- PackGate
- PackRec

2. Record

3. Reply

Epoch 0
Epoch 1

PackRec
PackGate
Normal operation

Epoch 0
- Execute
- Stop
- Check point

Epoch 1
- Execute
- Send state

1. Request
2. Record
3. Reply
4. ND Log

Primary
- Container
- Application

Backup
- PackGate
- PackRec
- RRC Agent
Normal operation

<table>
<thead>
<tr>
<th>Epoch 0</th>
<th>Epoch 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execute</td>
<td>Stop</td>
</tr>
<tr>
<td>Check point</td>
<td>Send state</td>
</tr>
</tbody>
</table>

1. Request

2. Record

3. Reply

4. ND Log

5. Release

- Application
- Container
- Primary
- Backup
- RRC Agent
- PackGate
- PackRec
Normal operation

1. Request

**Epoch 0**
- Execute
- Stop
- Check point

**Epoch 1**
- Execute
- Send state

2. Record
3. Reply
4. ND Log
5. Release
6. Reply

- RRC Agent
- PackGate
- PackRec

- Application
- Container
- Primary
- Backup
Normal operation

Epoch 0
- Execute
- Stop
- Check point

Epoch 1
- Execute
- Send state
- Check point

Application
- Record
- Reply

Container

Primary

Backup
- ND Log
- Release
- Reply

RRC Agent
- Request
- PackGate
- PackRec
Handling Primary Failure

Primary

Container

Application

Backup

PackRec

RRC Agent
Handling Primary Failure

- Application
- Container
- Primary

- RRC Agent
- PackRec
- Backup
Handling Primary Failure

RRC Agent

1. Restore

Application

PackRec

Backup

Container

Application

Primary

X
Handling Primary Failure

1. Restore
2. ND Log

RRC Agent

Application

PackRec

Backup

Recorded inputs
Handling Primary Failure

1. Restore
2. ND Log
3. Replay

Application
PackRec
RRC Agent
Backup
Recorded inputs

Primary
Container
Handling Primary Failure

1. Restore
2. ND Log
3. Replay
4. Request

Application

Container

Primary

PackRec

Backup
RRC: Backup Failure

Primary

Container

Application

Reply

PackGate

PackRec

Backup

Reply

Request

RRC Agent

Request

18
RRC: Backup Failure
RRC: Backup Failure

Request → Reply
Service IP

Application

Container

Primary

PackGate

PackRec

RRC Agent

Backup
RRC: Backup Failure

1. Service IP
2. Request

Primary

Container

Application

Reply

PackGate

PackRec

RRC Agent

Backup
RRC: Backup Failure

1. Service IP
2. Request
3. Reply

Application

Container

Primary

PackGate

PackRec

RRC Agent

Backup

18
Talk Outline

- Preface
- Motivation
- RRC overview
- Overcoming design and implementation challenges
- Evaluation
Key Design and Implementation Challenges

• Minimizing pause time during checkpointing
• Handling untracked nondeterministic events
• Robust integration of asynchronous checkpointing and recording of nondeterministic events
• Minimizing the overhead for collection and transfer of nondeterministic event logs
• Integration of TCP failover with replay during recovery
Key Design and Implementation Challenges

• Minimizing pause time during checkpointing
• Handling untracked nondeterministic events
• Robust integration of asynchronous checkpointing and recording of nondeterministic events
• Minimizing the overhead for collection and transfer of nondeterministic event logs
• Integration of TCP failover with replay during recovery
Service Pause during Container Checkpointing

Checkpointing requires saving a consistent state
→ Execution must pause during checkpointing
    → Service pause time during checkpointing
Service Pause during Container Checkpointing

Checkpointing requires saving a **consistent** state
→ Execution must pause during checkpointing
  → Service pause time during checkpointing

Container: tight state coupling with the underlying kernel
→ Significant in-kernel container state must be checkpointed
  → Retrieving the in-kernel container state is slow: **thousands of syscalls**
Service Pause during Container Checkpointing

Checkpointing requires saving a **consistent** state

→ Execution must pause during checkpointing
  → Service pause time during checkpointing

Container: tight state coupling with the underlying kernel

→ Significant in-kernel container state must be checkpointed
  → Retrieving the in-kernel container state is slow: **thousands of syscalls**

Checkpointing a container is **slow**
Service Pause during Container Checkpointing

Checkpointing requires saving a **consistent** state

→ Execution must pause during checkpointing
  → Service pause time during checkpointing

Container: tight state coupling with the underlying kernel

→ Significant in-kernel container state must be checkpointed
  → Retrieving the in-kernel container state is slow: **thousands of syscalls**

Checkpointing a container is **slow**

Challenge: minimize the pause time despite slow checkpointing
Minimizing Service Pause Using Container Fork

Key Idea: **Decouple** retrieval of in-kernel container state from container execution
Minimizing Service Pause Using Container Fork

Key Idea: **Decouple** retrieval of in-kernel container state from container execution

Design: New kernel primitive – Container fork
Minimizing Service Pause Using Container Fork

Key Idea: **Decouple** retrieval of in-kernel container state from container execution

Design: New kernel primitive – Container fork

**Primary Host**

- Application Container
Minimizing Service Pause Using Container Fork

Key Idea: **Decouple** retrieval of in-kernel container state from container execution

Design: New kernel primitive – Container fork

---

**Primary Host**

1. **Pause**

**Application Container**
Minimizing Service Pause Using Container Fork

Key Idea: **Decouple** retrieval of in-kernel container state from container execution

Design: New kernel primitive – Container fork

![Diagram](23)

1. Pause Application Container
2. Fork to Shadow Container
Minimizing Service Pause Using Container Fork

Key Idea: **Decouple** retrieval of in-kernel container state from container execution

Design: New kernel primitive – Container fork

![Diagram showing the process of minimizing service pause using container fork](image-url)

1. Resume
2. Fork
3. Application Container
4. Shadow Container

**Primary Host**
Minimizing Service Pause Using Container Fork

Key Idea: **Decouple** retrieval of in-kernel container state from container execution

Design: New kernel primitive – Container fork
Minimizing Service Pause Using Container Fork

Key Idea: **Decouple** retrieval of in-kernel container state from container execution

Design: New kernel primitive – Container fork

Result: Service Pause time [5.9ms - 42.9ms] $\rightarrow$ [0.5ms - 3.2ms]
RRC – Hybrid replication:
   Execution replay **only** during recovery
→ Vulnerability **only** to nondeterministic events occurring during
   the epoch of failure
Nondeterministic events and the Challenge of Data Races

RRC – Hybrid replication:
- Execution replay **only** during recovery
  → Vulnerability **only** to nondeterministic events occurring during the epoch of failure

RRC's handling of nondeterministic events:
- Replay nondeterministic event logs
Nondeterministic events and the Challenge of Data Races

RRC – Hybrid replication:
   - Execution replay **only** during recovery
   - Vulnerability **only** to nondeterministic events occurring during the epoch of failure

RRC's handling of nondeterministic events:
- Replay nondeterministic event logs

Multithreading: memory access ordering is nondeterministic

Solution:
- Record the order of all memory accesses
  - **Unacceptably high overhead**
Nondeterministic events and the Challenge of Data Races

RRC – Hybrid replication:
   Execution replay only during recovery
   → Vulnerability only to nondeterministic events occurring during the epoch of failure

RRC's handling of nondeterministic events:
   • Replay nondeterministic event logs

Multithreading: memory access ordering is nondeterministic

Solution:
   • Record the order of all memory accesses
     → Unacceptably high overhead
   • Record the outcomes of synchronization operations
     → Challenge: data races – unsynchronized memory accesses
Data Race Considerations

• Data races are **bugs**
• Impossible to eliminate **all** data races with languages like C/C++
Data Race Considerations

• Data races are **bugs**
• Impossible to eliminate **all** data races with languages like C/C++

• Existing tools can effectively detect frequently-manifested data races
• Deployed server applications go through testing / debugging
Data Race Considerations

• Data races are **bugs**
• Impossible to eliminate **all** data races with languages like C/C++

• Existing tools can effectively detect frequently-manifested data races
• Deployed server applications go through testing / debugging

→ RRC focuses on **infrequently-manifested** data races
The Potential Impact of Data Races

• During replay on the backup, most of system calls not actually executed
  → Significantly different timing of thread execution
The Potential Impact of Data Races

During replay on the backup, most of system calls not actually executed
→ Significantly different timing of thread execution
→ Outcomes of data races
→ Different outcomes of replay

Record Run
Thread1
Syscall1
Store X
Thread2
Syscall2
Load X

Replay Run
Thread1
Syscall1
Store X
Thread2
Syscall2
Load X
RRC’s Mitigation of the Impact of Data Races

- Record time intervals between system call returns on the primary
RRC’s Mitigation of the Impact of Data Races

- Record time intervals between system call returns on the primary
RRC’s Mitigation of the Impact of Data Races

- Record time intervals between system call returns on the primary
RRC’s Mitigation of the Impact of Data Races

- Record time intervals between system call returns on the primary
- Enforce inter-syscall interval during replay $\geq$ recorded interval
RRC’s Mitigation of the Impact of Data Races

- Record time intervals between system call returns on the primary
- Enforce inter-syscall interval during replay ≥ recorded interval

Recovery success rate with infrequent data races: \{35\%, 51\%\} \rightarrow 99\%
Talk Outline

• Preface
• Motivation
• RRC overview
• Overcoming design and implementation challenges
• Evaluation
Key design and implementation challenges

• Latency overhead
• Throughput overhead
• Recovery success rate
• Impact of data races
• CPU utilization overhead
• Pause time
• Recovery latency
• Impact of checkpoint interval
• Impact of workload footprint size and working set size
• Comparison with custom application-specific mechanisms
Key design and implementation challenges

• Latency overhead
• Throughput overhead
• Recovery success rate
  • Impact of data races
  • CPU utilization overhead
  • Pause time
  • Recovery latency
  • Impact of checkpoint interval
  • Impact of workload footprint size and working set size
  • Comparison with custom application-specific mechanisms
Evaluation Setup

• Baseline: NiLiCon: Container replication, checkpointing to a passive backup
Evaluation Setup

• Baseline: NiLiCon: Container replication, checkpointing to a passive backup

• Workloads:
  – In-memory databases: Redis, Tarantool, SSDB, Memcached, Aerospike
  – Webserver: Lighttpd
Evaluation Setup

• Baseline:
  NiLiCon: Container replication, checkpointing to a passive backup

• Workloads:
  – In-memory databases: Redis, Tarantool, SSDB, Memcached, Aerospike
  – Webserver: Lighttpd

• RRC configuration:
  – 100ms checkpointing interval
## Latency Overhead: RRC vs. NiLiCon

### Average Latency Overhead

<table>
<thead>
<tr>
<th></th>
<th>Lig</th>
<th>Redis</th>
<th>Taran</th>
<th>SSDB</th>
<th>Mem$</th>
<th>Aero</th>
</tr>
</thead>
<tbody>
<tr>
<td>RRC</td>
<td>144μs</td>
<td>198μs</td>
<td>211μs</td>
<td>263μs</td>
<td>169μs</td>
<td>290μs</td>
</tr>
<tr>
<td>NiLiCon</td>
<td>37ms</td>
<td>41ms</td>
<td>41ms</td>
<td>44ms</td>
<td>44ms</td>
<td>50ms</td>
</tr>
</tbody>
</table>
Latency Overhead: RRC vs. NiLiCon

**Average Latency Overhead**

<table>
<thead>
<tr>
<th></th>
<th>Lig</th>
<th>Redis</th>
<th>Taran</th>
<th>SSDB</th>
<th>Mem$</th>
<th>Aero</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>RRC</strong></td>
<td>144μs</td>
<td>198μs</td>
<td>211μs</td>
<td>263μs</td>
<td>169μs</td>
<td>290μs</td>
</tr>
<tr>
<td><strong>NiLiCon</strong></td>
<td>37ms</td>
<td>41ms</td>
<td>41ms</td>
<td>44ms</td>
<td>44ms</td>
<td>50ms</td>
</tr>
</tbody>
</table>

**Average:**  
RRC: 144μs – 290μs  
NiLiCon: 37ms – 50ms
## Latency Overhead: RRC vs. NiLiCon

### Average Latency Overhead

<table>
<thead>
<tr>
<th></th>
<th>Lig</th>
<th>Redis</th>
<th>Taran</th>
<th>SSDB</th>
<th>Mem$</th>
<th>Aero</th>
</tr>
</thead>
<tbody>
<tr>
<td>RRC</td>
<td>144μs</td>
<td>198μs</td>
<td>211μs</td>
<td>263μs</td>
<td>169μs</td>
<td>290μs</td>
</tr>
<tr>
<td>NiLiCon</td>
<td>37ms</td>
<td>41ms</td>
<td>41ms</td>
<td>44ms</td>
<td>44ms</td>
<td>50ms</td>
</tr>
</tbody>
</table>

Average: RRC: 144us – 290μs  
99th% : RRC: 235μs – 959μs  
NiLiCon: 37ms – 50ms  
NiLiCon: 39ms – 63ms
## Latency Overhead: RRC vs. NiLiCon

### Average Latency Overhead

<table>
<thead>
<tr>
<th></th>
<th>Lig</th>
<th>Redis</th>
<th>Taran</th>
<th>SSDB</th>
<th>Mem$</th>
<th>Aero</th>
</tr>
</thead>
<tbody>
<tr>
<td>RRC</td>
<td>144μs</td>
<td>198μs</td>
<td>211μs</td>
<td>263μs</td>
<td>169μs</td>
<td>290μs</td>
</tr>
<tr>
<td>NiLiCon</td>
<td>37ms</td>
<td>41ms</td>
<td>41ms</td>
<td>44ms</td>
<td>44ms</td>
<td>50ms</td>
</tr>
</tbody>
</table>

**Average:**
- RRC: 144μs – 290μs
- NiLiCon: 37ms – 50ms

**99th%:**
- RRC: 235μs – 959μs
- NiLiCon: 39ms – 63ms

RRC: Hybrid replication + Container fork ➔
- two orders of magnitude lower latency overhead
Throughput Overhead: RRC vs. NiLiCon

- **Lig**
  - NILI: 18%
  - RRC: 25%

- **Redis**
  - NILI: 35%
  - RRC: 46%

- **Taran**
  - NILI: 37%
  - RRC: 62%

- **SSDB**
  - NILI: 35%
  - RRC: 52%

- **Mem$**
  - NILI: 128%
  - RRC: 139%

- **Aero**
  - NILI: 85%
  - RRC: 100%

Legend:
- Green: Record ND events
- Blue: Pause for checkpointing
- Brown: Copy on write & Page Fault
Throughput Overhead: RRC vs. NiLiCon

RRC

NILI

Redis

NILI

RRC

Taran

NILI

RRC

SSDB

NILI

RRC

Mem$ (green)

NILI

RRC

Aero

NILI

RRC

Record ND events

Pause for checkpointing

Copy on write & Page Fault

Overhead (lower is better)
Throughput Overhead: RRC vs. NiLiCon

<table>
<thead>
<tr>
<th>Service</th>
<th>NILI Overhead</th>
<th>RRC Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lig</td>
<td>25%</td>
<td>18%</td>
</tr>
<tr>
<td>Redis</td>
<td>56%</td>
<td>46%</td>
</tr>
<tr>
<td>Taran</td>
<td>62%</td>
<td>37%</td>
</tr>
<tr>
<td>SSDB</td>
<td>62%</td>
<td>52%</td>
</tr>
<tr>
<td>Mem$</td>
<td>128%</td>
<td>35%</td>
</tr>
<tr>
<td>Aero</td>
<td>139%</td>
<td>85%</td>
</tr>
</tbody>
</table>

- **Lig**: 25% (NILI) vs. 18% (RRC)
- **Redis**: 56% (NILI) vs. 46% (RRC)
- **Taran**: 62% (NILI) vs. 37% (RRC)
- **SSDB**: 62% (NILI) vs. 52% (RRC)
- **Mem$**: 128% (NILI) vs. 35% (RRC)
- **Aero**: 139% (NILI) vs. 85% (RRC)

Legend:
- Green: Record ND events
- Blue: Pause for checkpointing
- Dark Brown: Copy on write & Page Fault

Overhead (lower is better)
Throughput Overhead: RRC vs. NiLiCon

- **Lig**
  - RRC: 18%
  - NILI: 25%
- **Redis**
  - RRC: 46%
  - NILI: 56%
- **Taran**
  - RRC: 37%
  - NILI: 62%
- **SSDB**
  - RRC: 52%
  - NILI: 62%
- **Mem$**
  - RRC: 35%
  - NILI: 128%
- **Aero**
  - RRC: 85%
  - NILI: 139%

Legend:
- Green: Record ND events
- Blue: Pause for checkpointing
- Black: Copy on write & Page Fault

Overhead (lower is better)
Throughput Overhead: RRC vs. NiLiCon

- **Lig**
  - NILI: 25%
  - RRC: 18%

- **Redis**
  - NILI: 56%
  - RRC: 46%

- **Taran**
  - NILI: 62%
  - RRC: 37%

- **SSDB**
  - NILI: 62%
  - RRC: 52%

- **Mem$**
  - NILI: 25%
  - RRC: 56%

- **Aero**
  - NILI: 62%
  - RRC: 85%

Overhead (lower is better)

- **Record ND events**
- **Pause for checkpointing**
- **Copy on write & Page Fault**

Overhead values:
- **Lig**: NILI 25%, RRC 18%
- **Redis**: NILI 56%, RRC 46%
- **Taran**: NILI 62%, RRC 37%
- **SSDB**: NILI 62%, RRC 52%
- **Mem$**: NILI 25%, RRC 56%
- **Aero**: NILI 62%, RRC 85%
Throughput Overhead: RRC vs. NiLiCon

- **Lig**
  - NILI: 25%
  - RRC: 18%

- **Redis**
  - NILI: 56%
  - RRC: 46%

- **Taran**
  - NILI: 37%
  - RRC: 62%

- **SSDB**
  - NILI: 62%
  - RRC: 52%

- **Mem$**
  - NILI: 35%
  - RRC: 128%

- **Aero**
  - NILI: 85%
  - RRC: 139%

Legend:
- Green: Record ND events
- Blue: Pause for checkpointing
- Dark Green: Copy on write & Page Fault

Overhead (lower is better)
Throughput Overhead: RRC vs. NiLiCon

<table>
<thead>
<tr>
<th>System</th>
<th>Recording overhead:</th>
<th>RRC: 14% - 47%</th>
<th>NiLiCon: 0%</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lig</td>
<td>RRC: 18%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Redis</td>
<td>RRC: 46%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Taran</td>
<td>RRC: 37%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SSDB</td>
<td>RRC: 52%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Mem$</td>
<td>RRC: 35%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Aero</td>
<td>RRC: 85%</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Record ND events**
- **Pause for checkpointing**
- **Copy-on-write & Page Fault**
Throughput Overhead: RRC vs. NiLiCon

- **Lig**
  - RRC: 18%
  - NiLiCon: 25%

- **Redis**
  - RRC: 46%
  - NiLiCon: 56%

- **Taran**
  - RRC: 37%
  - NiLiCon: 62%

- **SSDB**
  - RRC: 130%
  - NiLiCon: 52%

- **Mem$**
  - RRC: 128%

- **Aero**
  - RRC: 139%

**Recording overhead:**
- RRC: 14% - 47%
- NiLiCon: 0%

**Pause overhead:**
- RRC: 1% - 3%
- NiLiCon: 17% - 130%
Throughput Overhead: RRC vs. NiLiCon

**Lig**
- RRC: 18%
- NiLiCon: 25%

**Redis**
- RRC: 46%
- NiLiCon: 56%

**Taran**
- RRC: 37%
- NiLiCon: 62%

**SSDB**
- RRC: 130%
- NiLiCon: 62%

**Mem$**
- RRC: 35%
- NiLiCon: 128%

**Aero**
- RRC: 85%
- NiLiCon: 139%

**Recording overhead:**
- RRC: 14% - 47%
- NiLiCon: 0%

**Pause overhead:**
- RRC: 1% - 3%
- NiLiCon: 17% - 130%

**Overall:**
- RRC: 18% - 85%
- NiLiCon: 25% - 139%
Recovery Success Rate

Fault injection setups:
• Fail-stop failures
• 1000s of fault injections
• Injection into both the primary and the backup host
Recovery Success Rate

Fault injection setups:

- Fail-stop failures
- 1000s of fault injections
- Injection into both the primary and the backup host

Recovery rate:

- >99% with real-world examples of data races
- **100%** without data races
Summary

• *Key goals:* Application-transparent fault tolerance for server applications
  – Multithreading
  – Minimize latency and throughput overhead
Summary

• **Key goals:** Application-transparent fault tolerance for server applications
  – Multithreading
  – Minimize latency and throughput overhead

• **Key insight:** **decouple** replication-related operations from normal operations
  – checkpoint interval ↔ delay in releasing outputs
  – time to take a checkpoint ↔ service interruption
  – Untracked nondeterminism ↔ service interruption
Summary

• **Key goals:** Application-transparent fault tolerance for server applications
  – Multithreading
  – Minimize latency and throughput overhead

• **Key insight:** *decouple* replication-related operations from normal operations
  – checkpoint interval ↔ delay in releasing outputs
  – time to take a checkpoint ↔ service interruption
  – Untracked nondeterminism ↔ service interruption

• **Key mechanisms:** hybrid replication: checkpointing + deterministic replay
  container fork
  passive backup
  mitigation of the impact of data races
Summary

- **Key goals:** Application-transparent fault tolerance for server applications
  - Multithreading
  - Minimize latency and throughput overhead

- **Key insight:** **decouple** replication-related operations from normal operations
  - checkpoint interval ↔ delay in releasing outputs
  - time to take a checkpoint ↔ service interruption
  - Untracked nondeterminism ↔ service interruption

- **Key mechanisms:** hybrid replication: checkpointing + deterministic replay
  - container fork
  - passive backup
  - mitigation of the impact of data races

- **Key results:**
  - average latency overhead < 290us vs. 10s of ms with passive backup
  - throughput overhead < 85% vs. < 139% with passive backup
  - recovery rate for fail-stop failures:
    - >99% with real-world examples of data races
    - 100% without data races
Support for Deterministic Replay

Requirement:
• Record nondeterministic events on the primary
• Transfer the log to the backup
• Replay the log for recovery on the backup

Nondeterministic events:
• External inputs – e.g., network packets from the clients
• Synchronization operations – e.g., lock acquire/release
• Certain local operations -- e.g., gettimeofday()