Design and Modeling of a Non-blocking Checkpointing System

Kento Sato[†1,2], Adam Moody[†3], Kathryn Mohror[†3], Todd Gamblin[†3],
Bronis R. de Supinski[†3], Naoya Maruyama[†4,5] and Satoshi Matsuoka[†1,5,6,7]

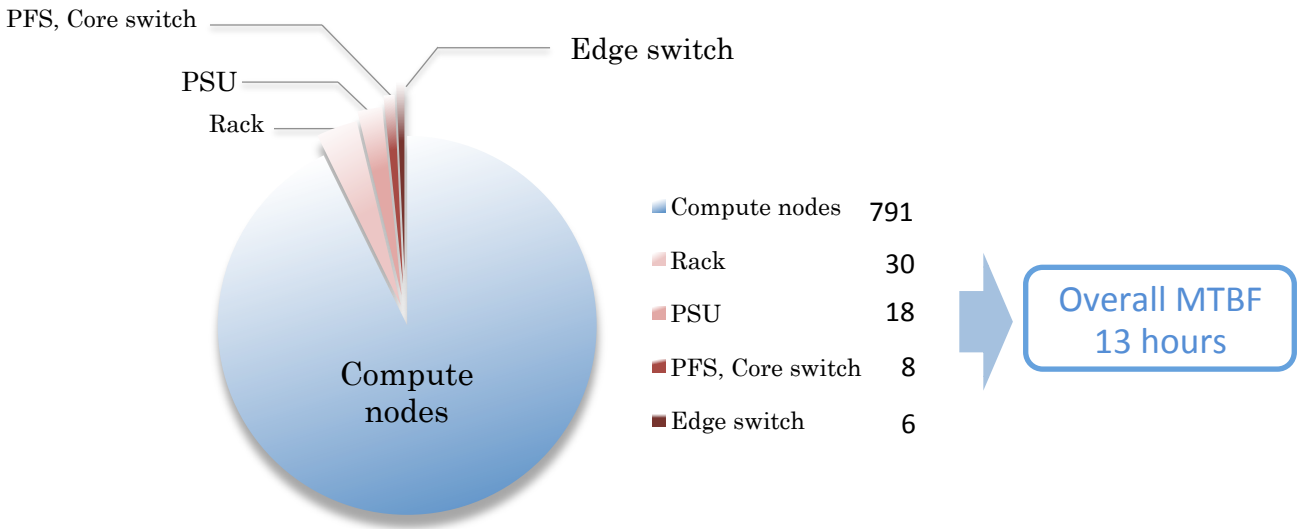
†1 Tokyo Institute of Technology
†2 Research Fellow of the Japan Society for the Promotion of Science
†3 Lawrence Livermore National Laboratory
†4 RIKEN Advanced institute for Computational Science
†5 Global Scientific information and Computing Center
†6 National Institute of Informatics
†7 JST/CREST

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52- 07NA27344. LLNL-PRES-599833-DRAFT

November 13th, 2012

Failures on HPC systems

- Exponential growth in computational power
  - Enables finer grained scientific simulations
- Overall failures rate increases accordingly
  - Due to increasing complexity and system size



Failure analysis on TSUBAME2.0

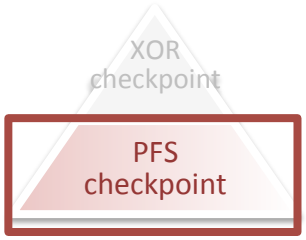
Period: 1.5 years (Nov 1st, 2010 ~ April 6th 2012)
Observations: 962 node failures in total

TSUBAME2.0, 14th in Top500 (June 2012)

2.4 PFlops
1442 nodes
2953 CPU sockets
4264 GPUs
197 switches
58 racks

- System resiliency is becoming more important
  - Without a viable resilience strategy, applications can not run for even one day on such a large system

Traditional Checkpoint/Restart

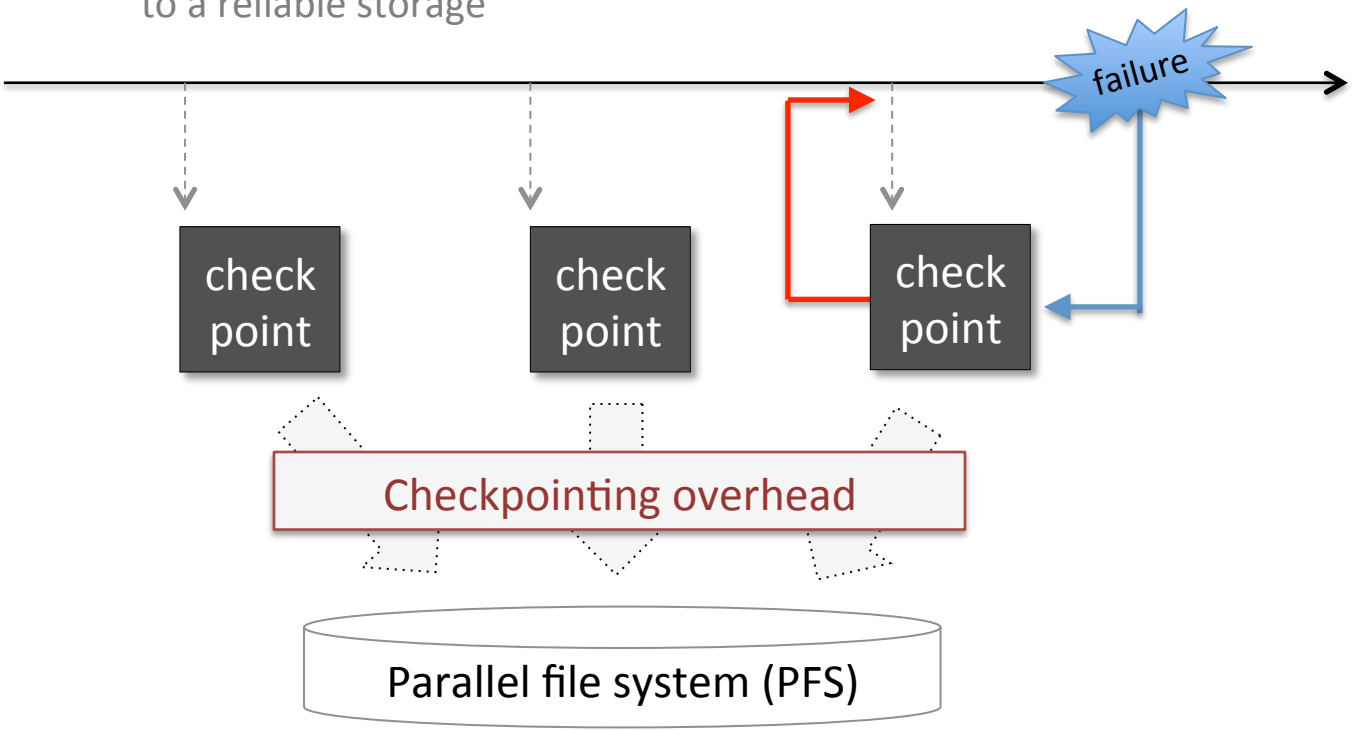


Checkpoint

Periodically save a snapshot of an application state to a reliable storage

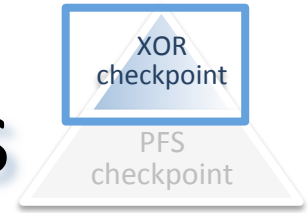
Restart

On a failure, restart the execution from the latest checkpoint

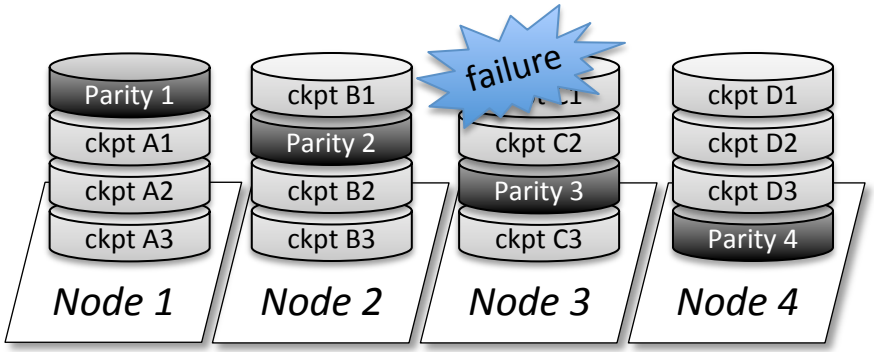


Mostly these checkpoints are stored in the most reliable storage, such as a shared parallel file system(PFS).

Scalable checkpointing methods



- Diskless checkpoint:
  - Create redundant data across local storages on compute nodes using a encoding technique such as XOR
  - Can restore lost checkpoints on a failure caused by small # of nodes like RAID-5

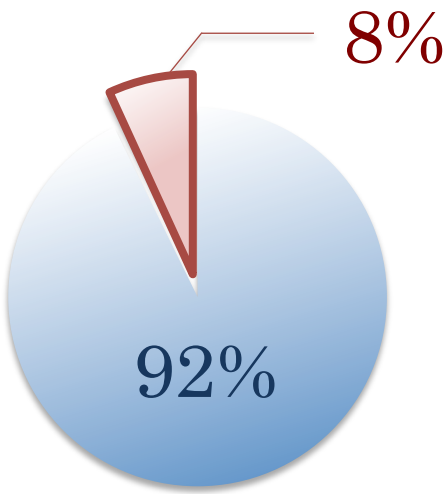


XOR encoding example

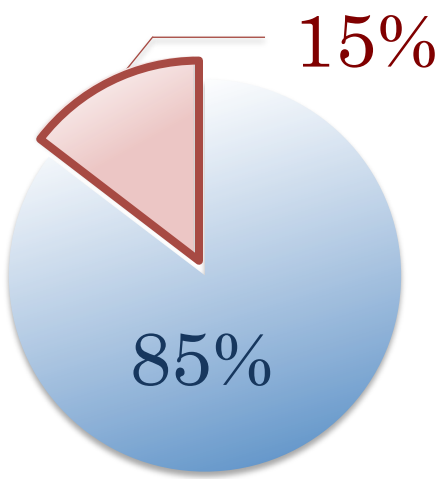
- Most of failures comes from one node, or can recover from XOR checkpoint
  - e.g. 1) TSUBAME2.0: 92% failures
  - e.g. 2) LLNL clusters: 85% failures

Rest of failures still require a checkpoint on a reliable PFS

Diskless checkpoint is promising approach

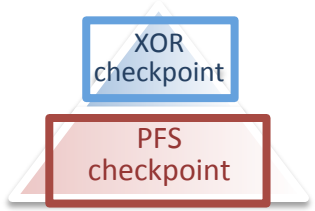


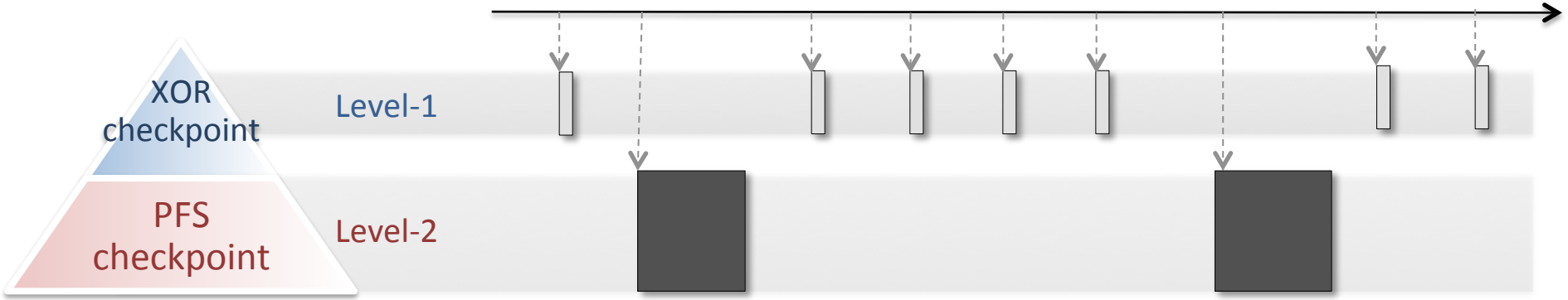
Failure analysis on TSUBAME2.0



Failure analysis on LLNL clusters

Multi-level checkpointing (MLC)





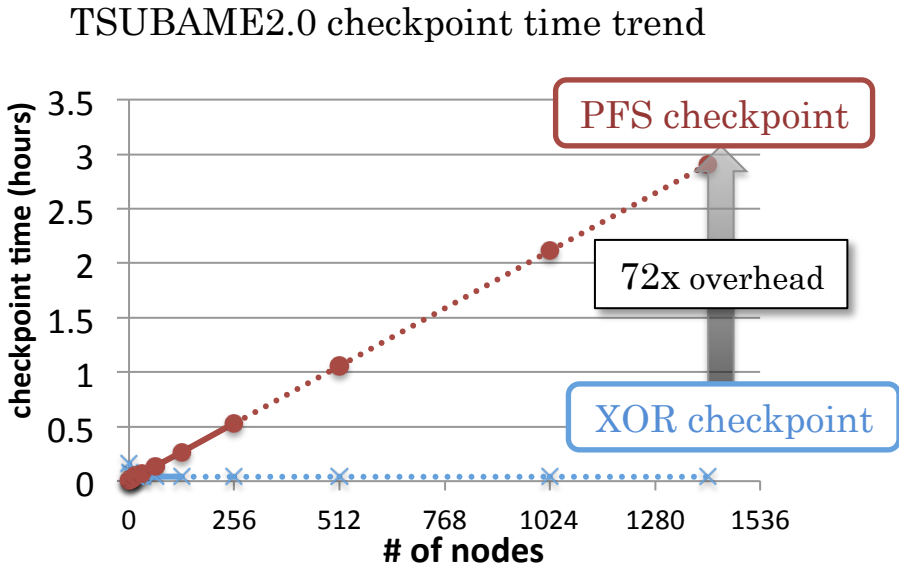
- Use storage levels hierarchically
  - XOR checkpoint: Frequently
    - for one node or a few node failure
  - PFS checkpoint: Less frequently
    - for multi-node failure
- 8x efficiency improvement
  - With MLC implementation called SCR(Scalable Checkpoint/Restart) library developed in LLNL
  - Compared to single-level checkpointing

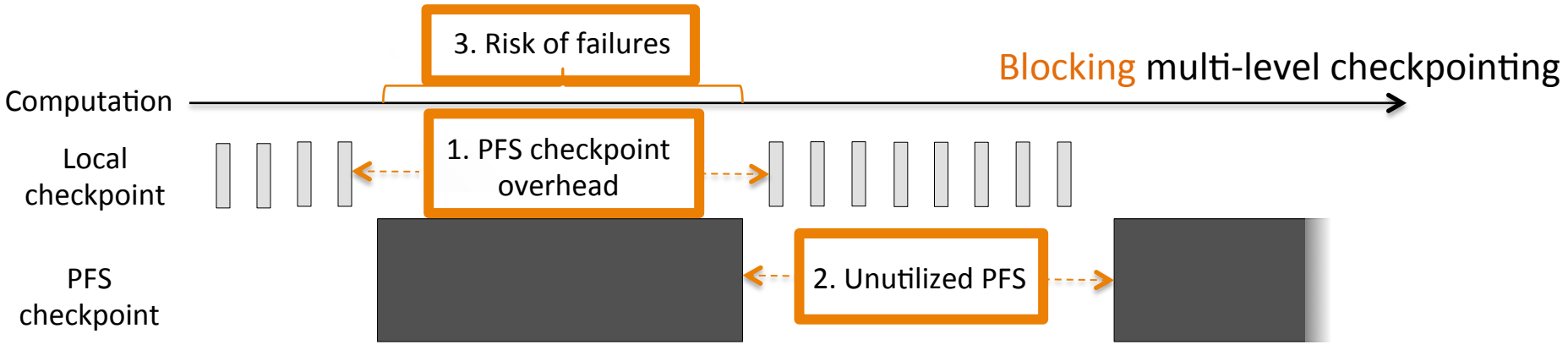
Source: A. Moody, G. Bronevetsky, K. Mohror, and B. R. de Supinski, "Design, Modeling, and Evaluation of a Scalable Multi-level Checkpointing System," in Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC 10).

MLC Problems on Petascale or larger

Three potential problems

1. PFS checkpoint overhead
   - Even with MLC, PFS checkpoint still becomes big overhead
2. Inefficient PFS utilization
   - Time between PFS checkpoints becomes long, PFS is not utilized during XOR checkpoints
3. Failure during PFS checkpoint
   - At scale, prolonged PFS checkpointing has a risk of failures during checkpointing





Objective, Proposal and Contributions

- **Objective**: More efficient MLC
  - Minimize PFS checkpoint overhead
  - Improve PFS utilization
  - Reduce a risk of failure during PFS checkpoint

- **Proposal & Contributions**:
  - Developed an non-blocking checkpointing system as an extension for SCR library
    - PFS checkpoint with 0.5 ~ 2.5% overhead
  - Modeled the non-blocking checkpointing
    - Determine optimal multi-level checkpoint configuration
    - 1.1 ~ 1.8x efficiency on current and future systems

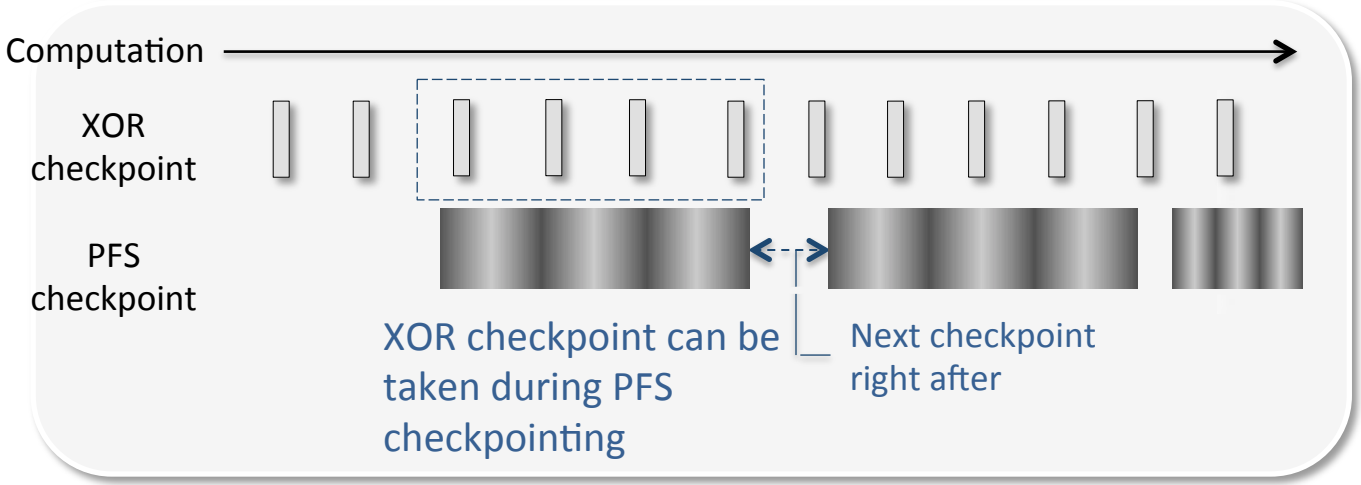
Outline

- Introduction
- Design of a Non-blocking checkpointing system
- Modeling of the Non-blocking checkpointing
- Evaluation
- Summary

Non-blocking checkpointing overview

Blocking multi-level checkpointing

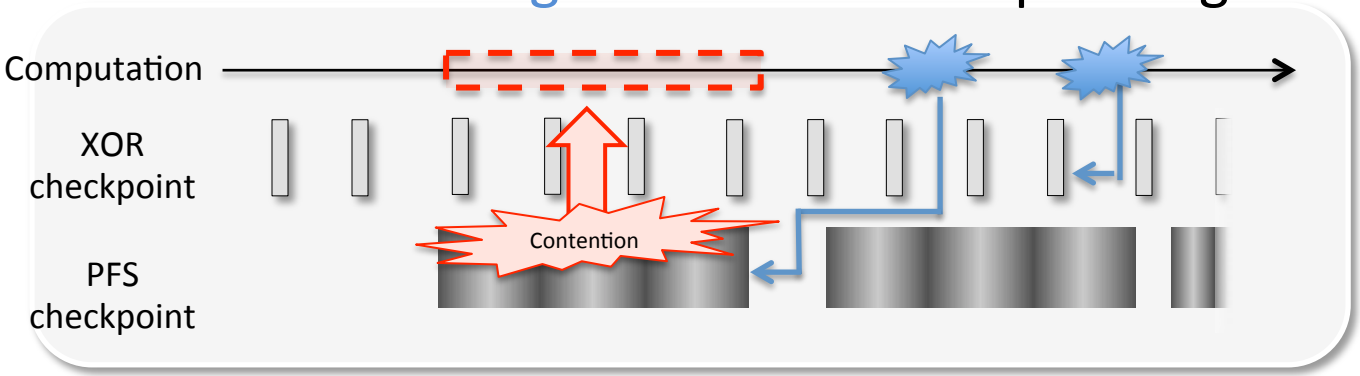
Non-blocking multi-level checkpointing



- Write PFS checkpoint in the background, minimize overhead
- By initiating next ckpt right after previous one, increase utilization
- Reduce impact of failures requiring XOR checkpoint

Challenges on Non-blocking checkpointing

Non-blocking multi-level checkpointing



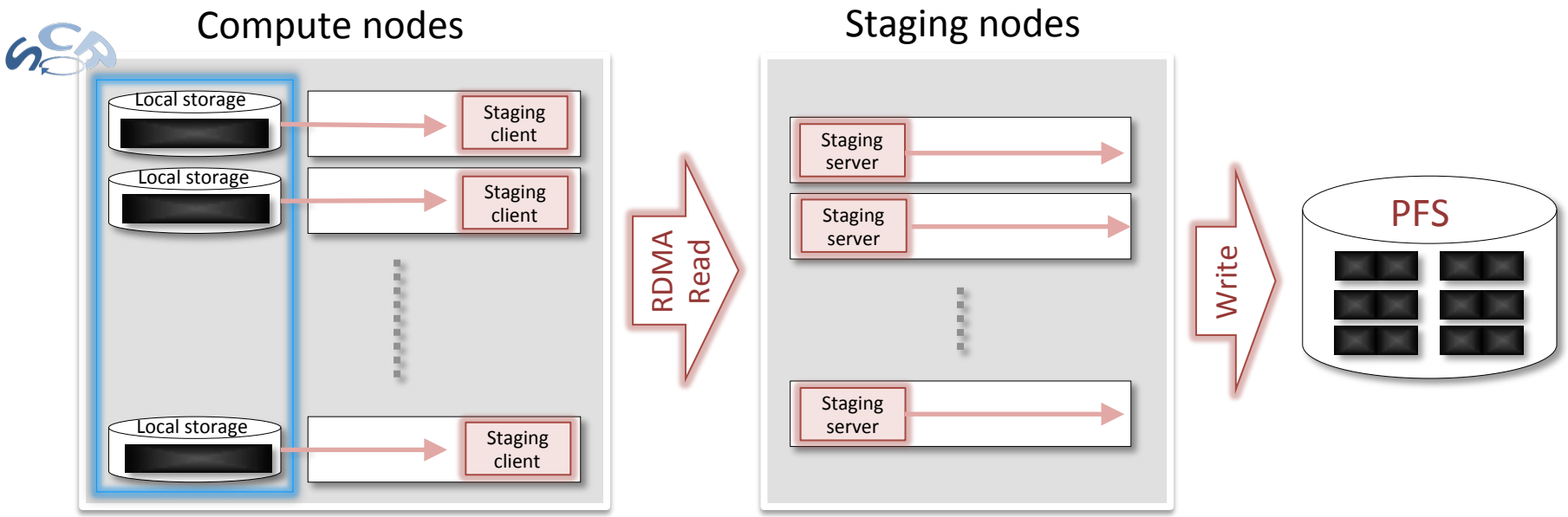
- Utilize local SSDs for the additional space
  - Write PFS checkpoint in the background which requires additional storage spaces
- Minimize resource contention
  - PFS checkpointing is running in the background, inflate the runtime due to resource contention

  ⇒ Implementation: Use RDMA with checkpoint dedicated nodes
- Optimize configuration (e.g. checkpoint interval)
  - On a failure requiring PFS , need "complete PFS checkpoint"
  - On a failure requiring XOR, need to restore both XOR & PFS ckpt being written

  ⇒ Modeling: Model a non-blocking multi-level checkpoint

Non-blocking checkpointing overview

- Between compute nodes and PFS, use staging nodes
  - Dedicated extra nodes for transferring local checkpoints written by a SCR library
  - Read checkpoints from compute nodes using RDMA, write out to a PFS

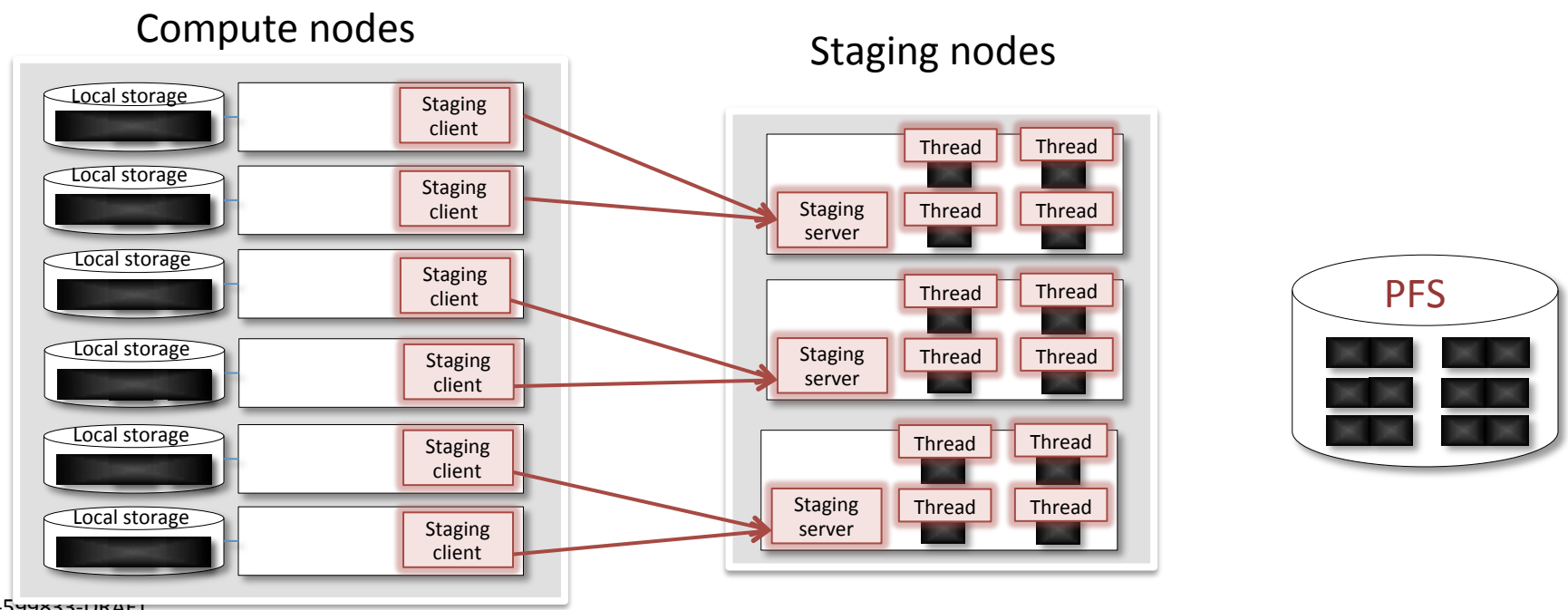


Local checkpoint

PFS checkpoint

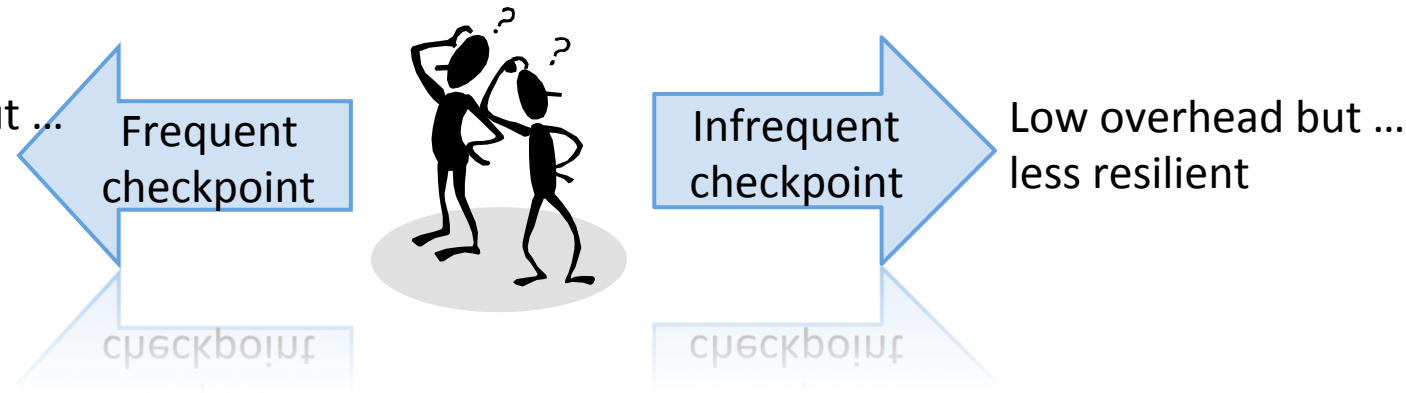
Non-blocking checkpointing using RDMA

1. Local storages to Local memory
   - After SCR writes checkpoint to a local storage, staging clients running on compute nodes read chunks of the checkpoint from the local storage to a buffer memory
2. Local memory to Remote memory
   - Send RDMA Read requests to a mapped staging server running on a staging node, staging server read the checkpoints from the buffer using RDMA
3. Remote memory to PFS
   - Data writer threads running on Staging nodes write checkpoint chunks to PFS in parallel



Modeling of Non-blocking checkpoint

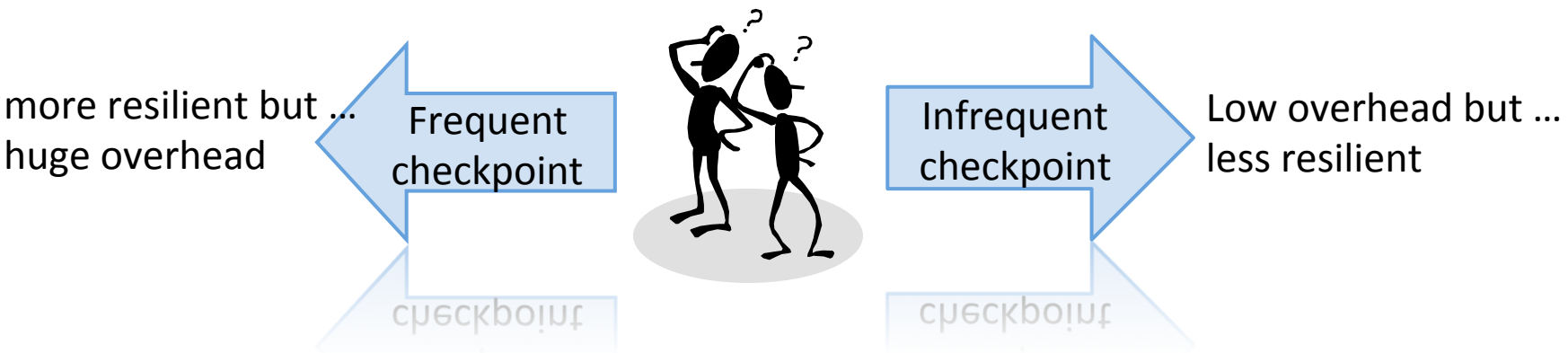
more resilient but …
huge overhead



Low overhead but …
less resilient

Outline

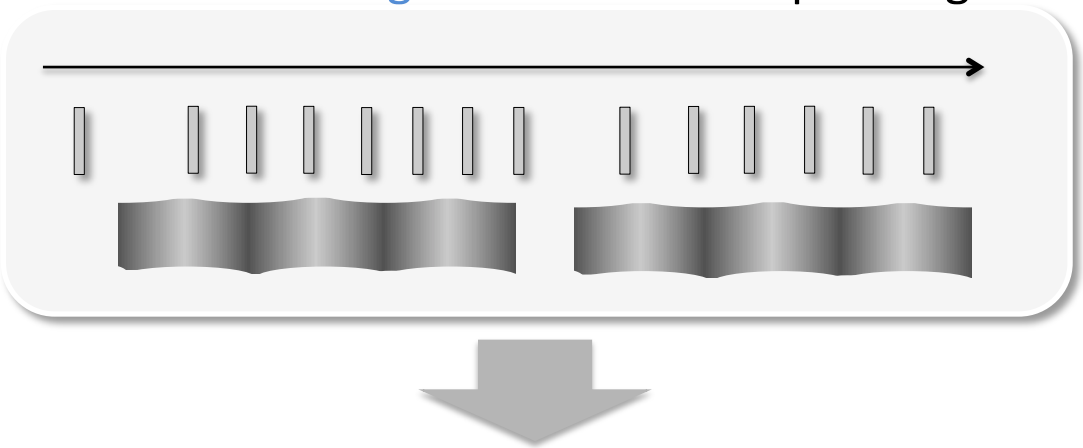
- Introduction
- Design of Non-blocking checkpointing system
- Modeling of Non-blocking checkpointing
- Evaluation
- Summary



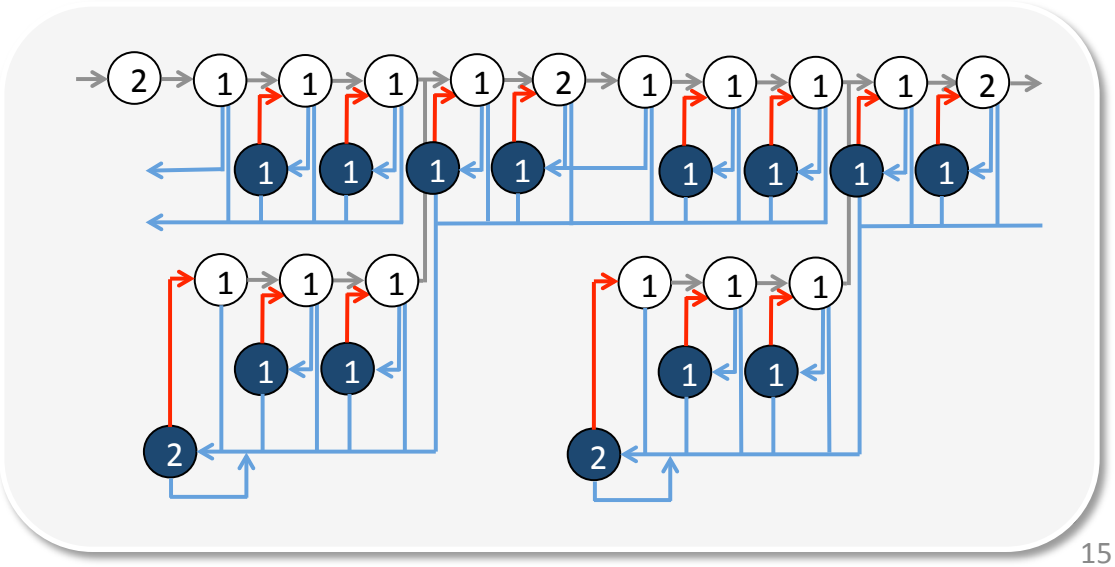
Non-blocking MLC model overview

- Describe an application's state transitions as Markov model
- Input (each level of ..)
  - Checkpoint time
  - Restart time
  - Failure rate
  - Interval
- Output
  - Expected runtime
- Find checkpoint intervals that minimize runtime

Non-blocking multi-level checkpointing

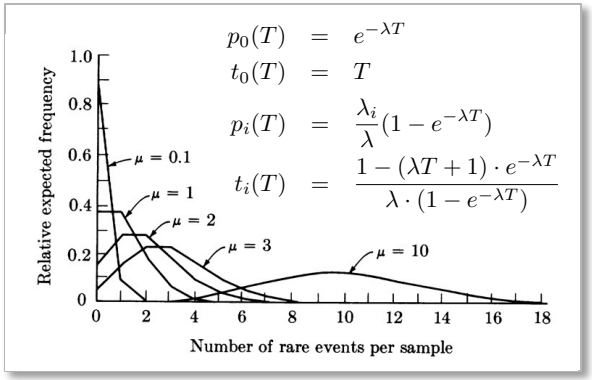


Non-blocking multi-level checkpoint model



Assumptions on the model

- Independent and identically distributed failure rate & Poisson distribution
  - One failure does not increase the probability of successive failures



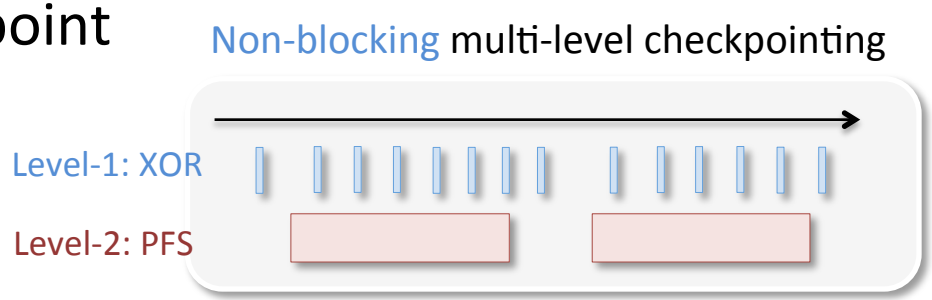
- Stable write & read performance
  - Checkpoint/Restart time significantly does not change during overall the runtime

- Failure on Level-k recovery => Level-(k+1) checkpoint
  - Another one node failure during XOR recovery requires a PFS checkpoint
  - Assume PFS checkpoint can retry infinitely

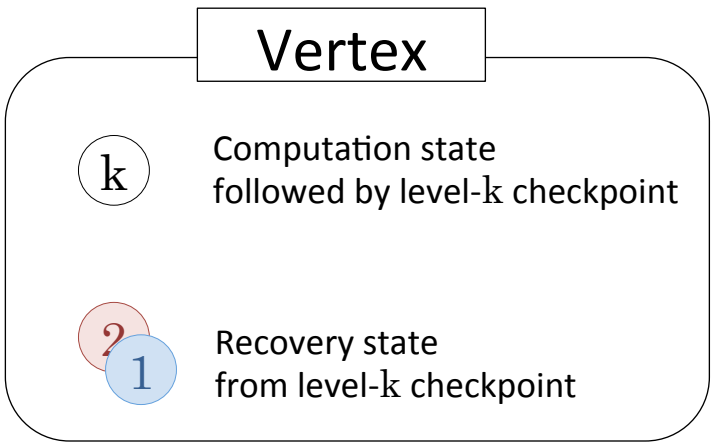
- Saved checkpoints are never lost on non-failed nodes and a PFS
  - Guarantee failed job can restart from the latest checkpoint

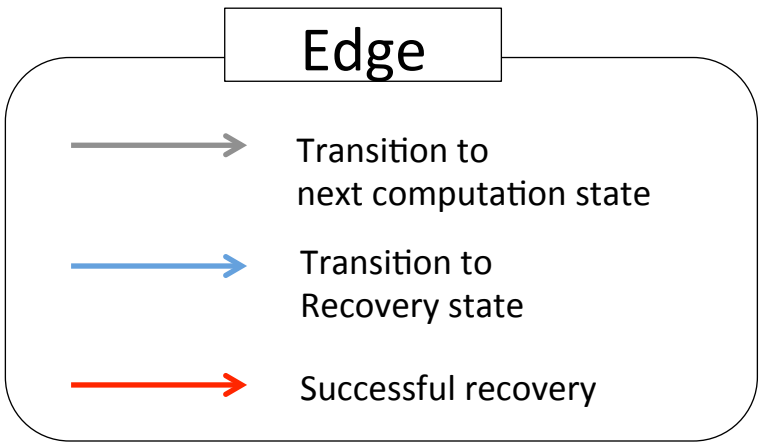
Two-level checkpoint example

- For simplicity, two-level checkpoint
  - Level-1: XOR checkpoint
  - Level-2: PFS checkpoint

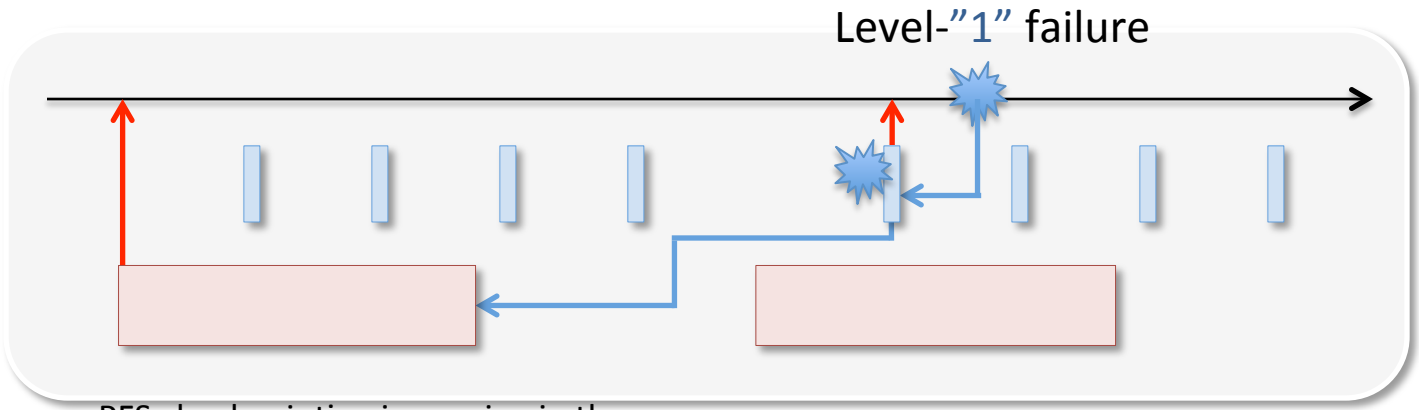


- Describe state transitions as Markov model



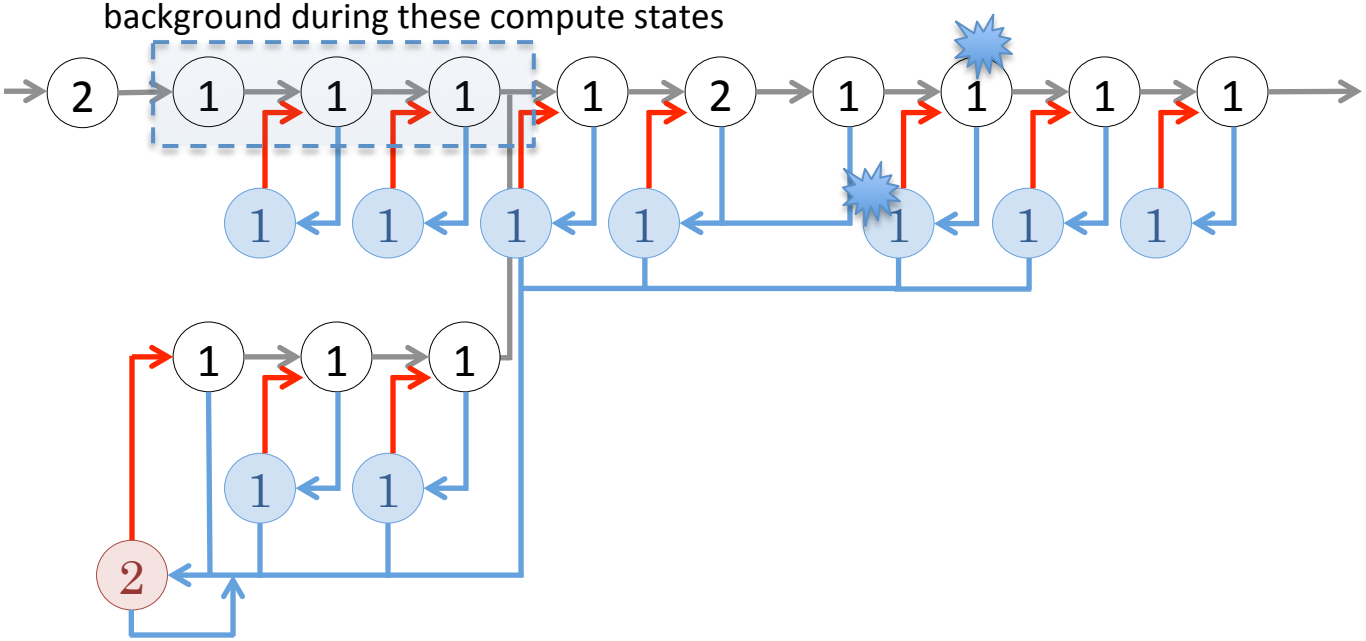


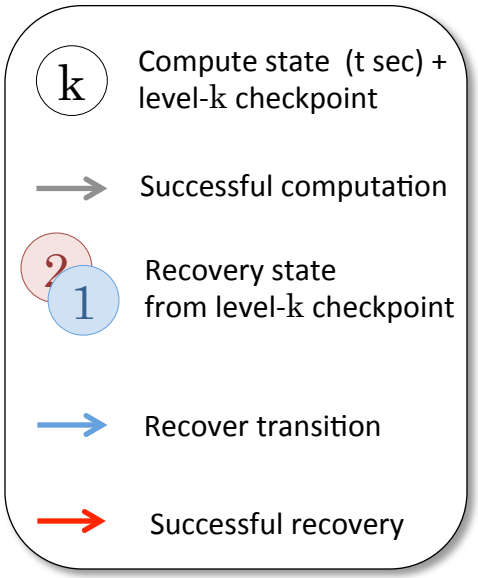
No failure & Level-"1" failure case



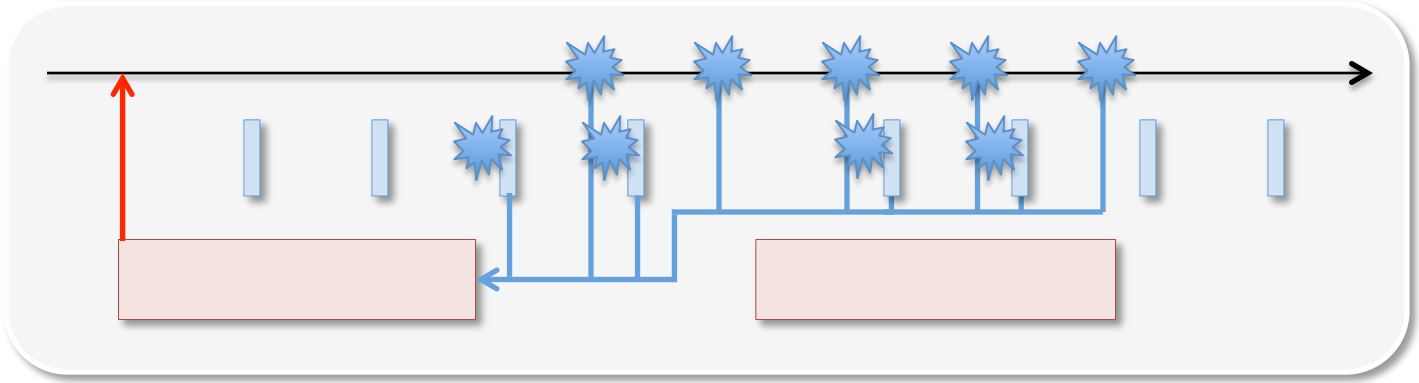
L-2 ckpt: L-1 ckpt
= 1: 4

PFS checkpointing is running in the background during these compute states

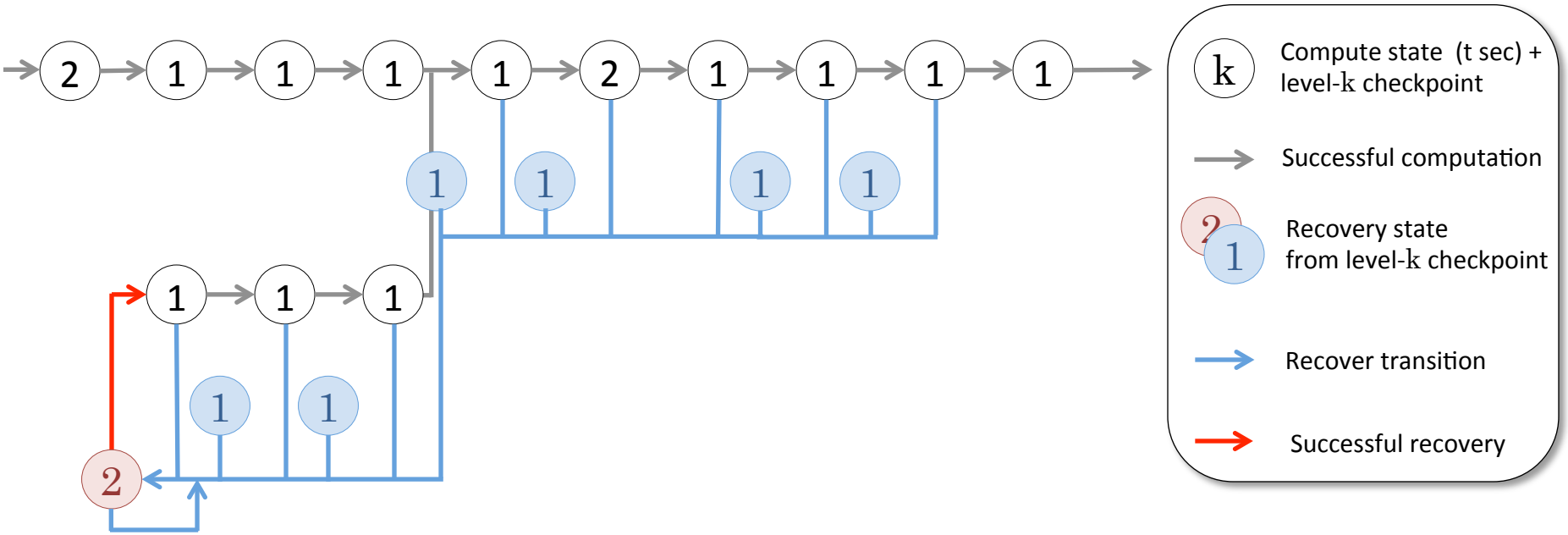




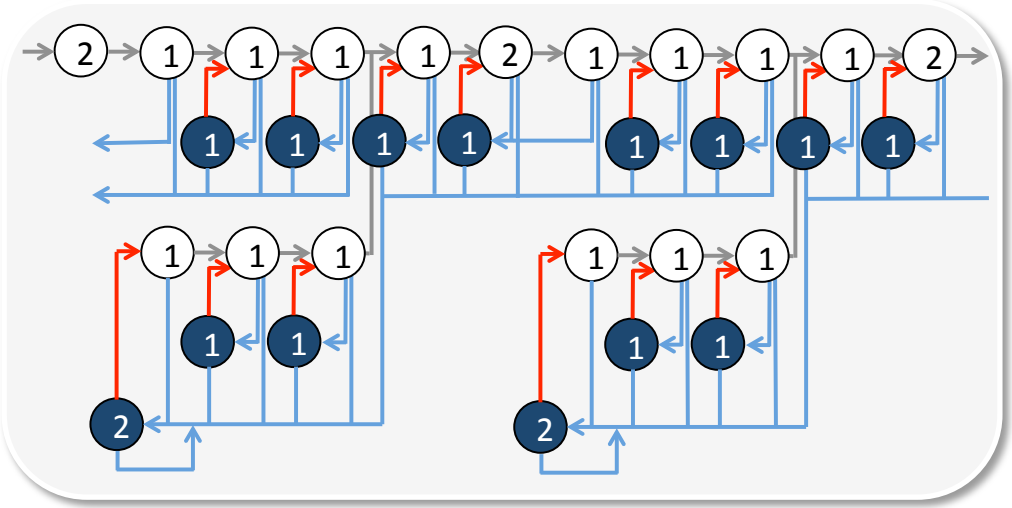
Level-"2" failure case



L-2 ckpt: L-1 ckpt
= 1: 4



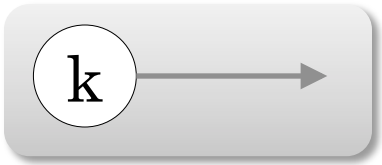
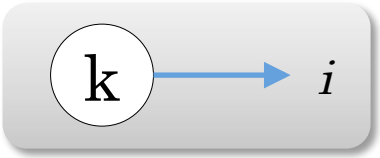
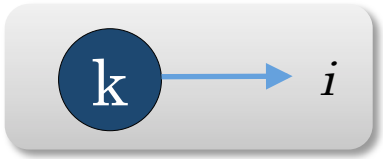
How to calculate *expected_runtime* ?



t : Interval

c_c : c-level checkpoint time

r_c : c-level recovery time

	Duration $t + c_k$	Duration r_k
No failure	 $p_0(t+c_k)$ $t_0(t+c_k)$	 $p_0(r_k)$ $t_0(r_k)$
Failure	 $p_i(t+c_k)$ $t_i(t+c_k)$	 $p_i(r_k)$ $t_i(r_k)$

$$
\begin{aligned}
p_0(T) &= e^{-\lambda T} \\
t_0(T) &= T \\
p_i(T) &= \frac{\lambda_i}{\lambda}(1 - e^{-\lambda T}) \\
t_i(T) &= \frac{1 - (\lambda T + 1) \cdot e^{-\lambda T}}{\lambda \cdot (1 - e^{-\lambda T})}
\end{aligned}
$$

λ_i : i-level checkpoint time

$\lambda = \sum \lambda_i$

$p_0(T)$: No failure for T seconds

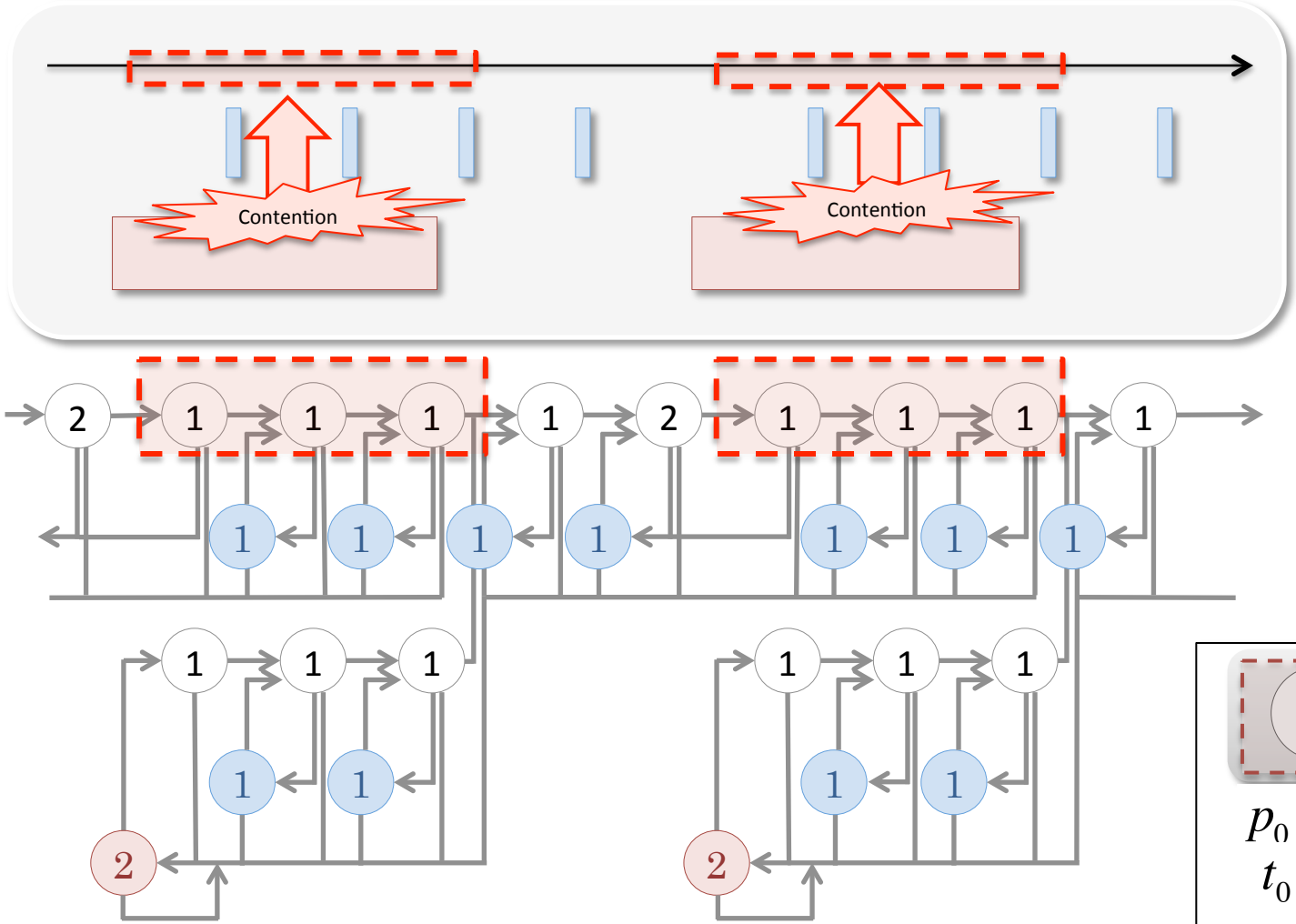
$t_0(T)$: Expected time when $p_0(T)$

$p_i(T)$: i-level failure for T seconds

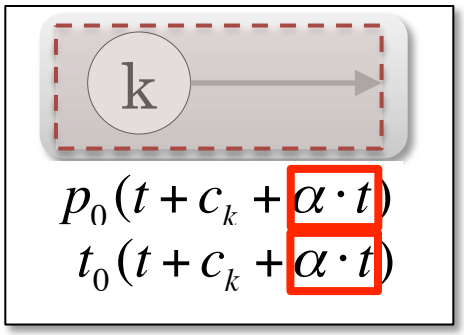
$t_i(T)$: Expected time when $p_i(T)$

Overhead factor: α

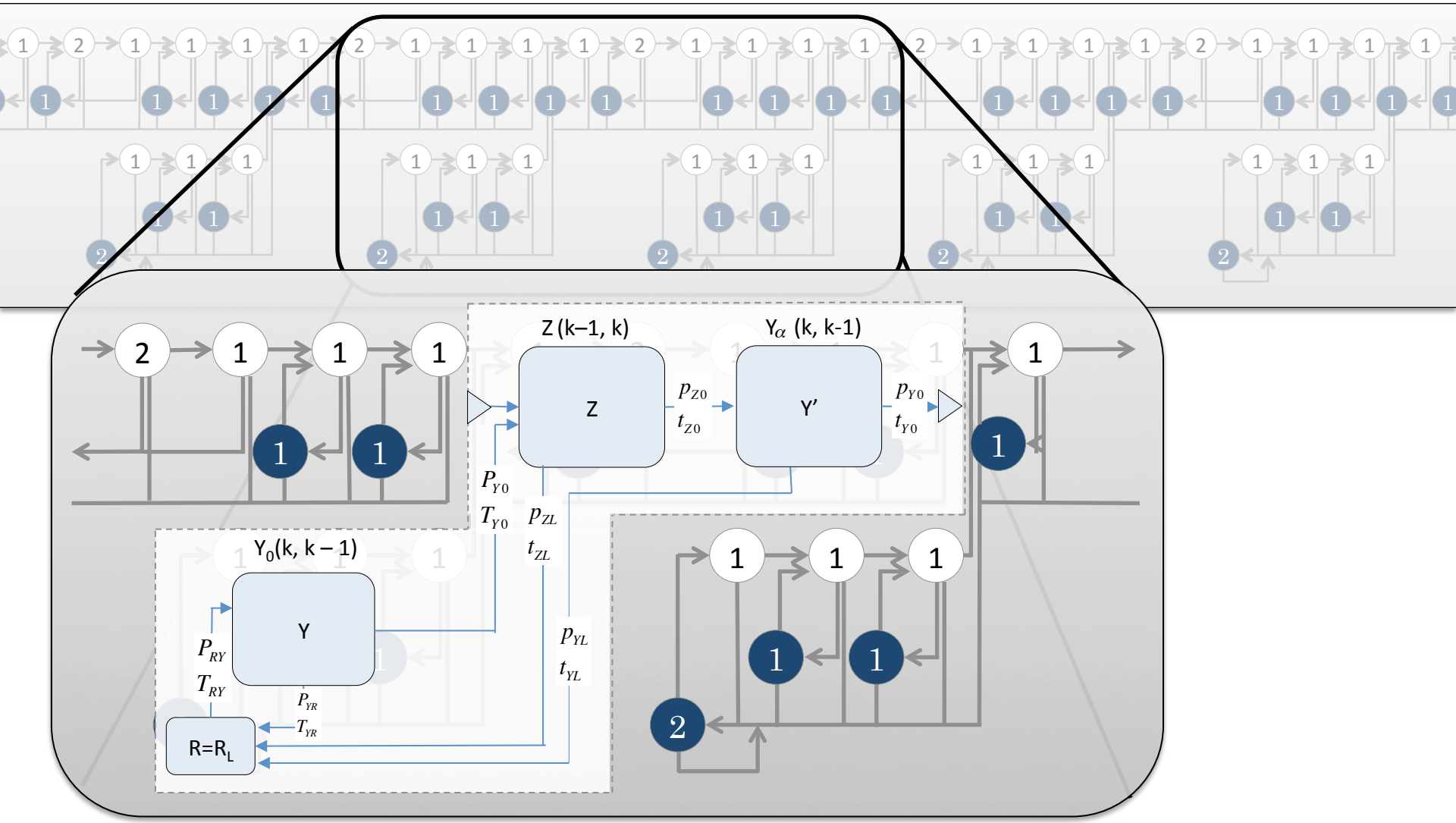
- Quantify an overhead by our proposed non-blocking checkpointing system



During these compute states PFS checkpointing is running in the background, inflate the runtime due to resource contention



Arbitrary N - level checkpointing model



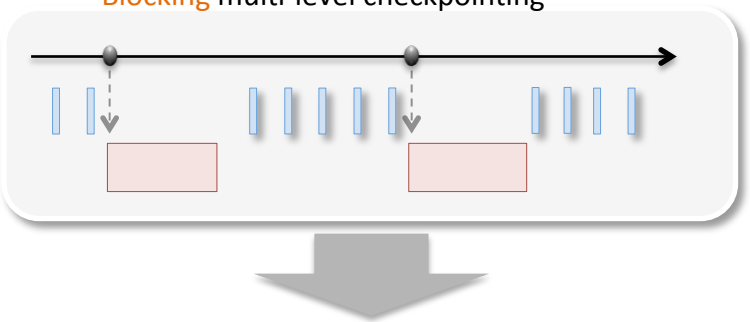
Non-blocking vs. Blocking MLC checkpointing

- **Benchmark**: Himeno benchmark
  - Stencil application solving Poisson's equation using Jacobi iteration method
- **Target System**:
  TSUBAME2.0 Thin nodes (1408 nodes)

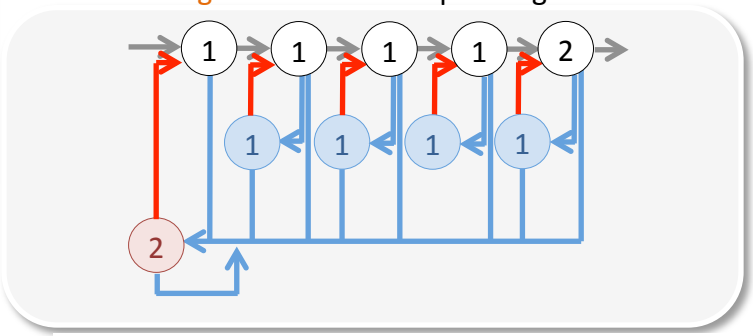
CPU	Intel Xeon X5670 2.93GHz (6cores x 2 sockets)
Memory	DDR3 1333MHz (58GB)
Network	Mellanox Technologie Dual rail QDR Infiniband 4x (80Gbps)
Local storage	120GB Intel SSD (RAID0/60GBx2)
PFS	Lustre (/work0)

- **Checkpoint Level**: Two-level
  - Level-1: XOR using local SSD
  - Level-2: PFS using Lustre

Blocking multi-level checkpointing



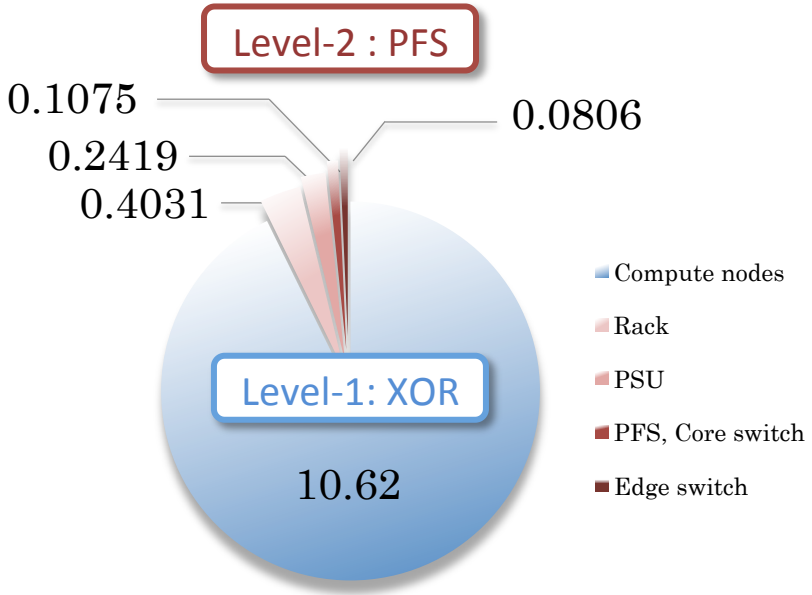
Blocking multi-level checkpointing model



Source: A. Moody, G. Bronevetsky, K. Mohror, and B. R. de Supinski, "Design, Modeling, and Evaluation of a Scalable Multi-level Checkpointing System," in Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC 10).

Model Parameters

- **Failure rates**
  - 1.5 years (Nov 1st 2010 ~ Apr 6th 2012) failure history

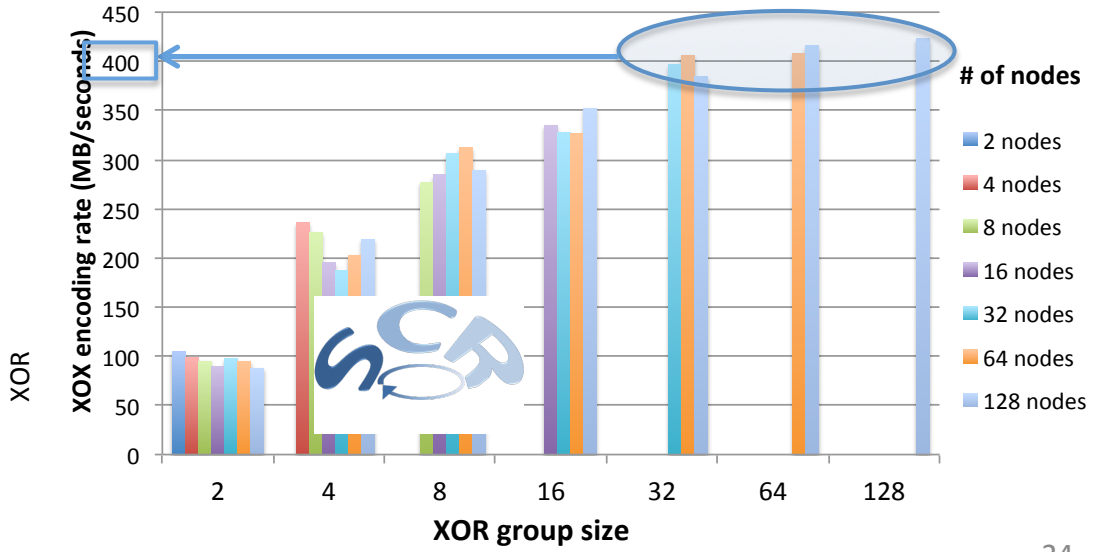


Failure rates (failures/week) on TSUBAME2.0

- **Checkpoint size per node**: 29GB
  - TSUBAME nodes memory: 58GB

Level-1

- **XOR throughput**: 400MB/s

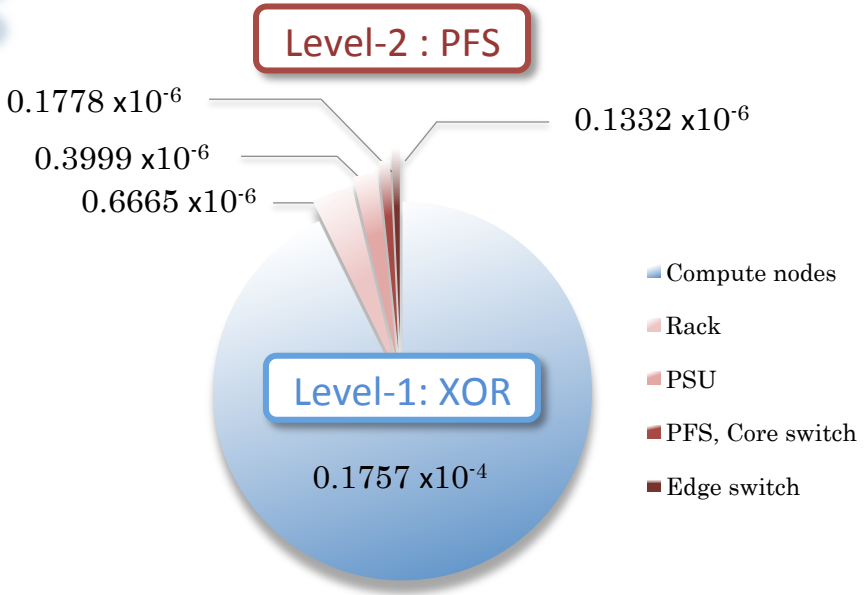


XOR encoding performance on TSUBAME2.0 using local SSDs

Model Parameters

- **Failure rates**
  - 1.5 years (Nov 1st 2010 ~ Apr 6th 2012) failure history

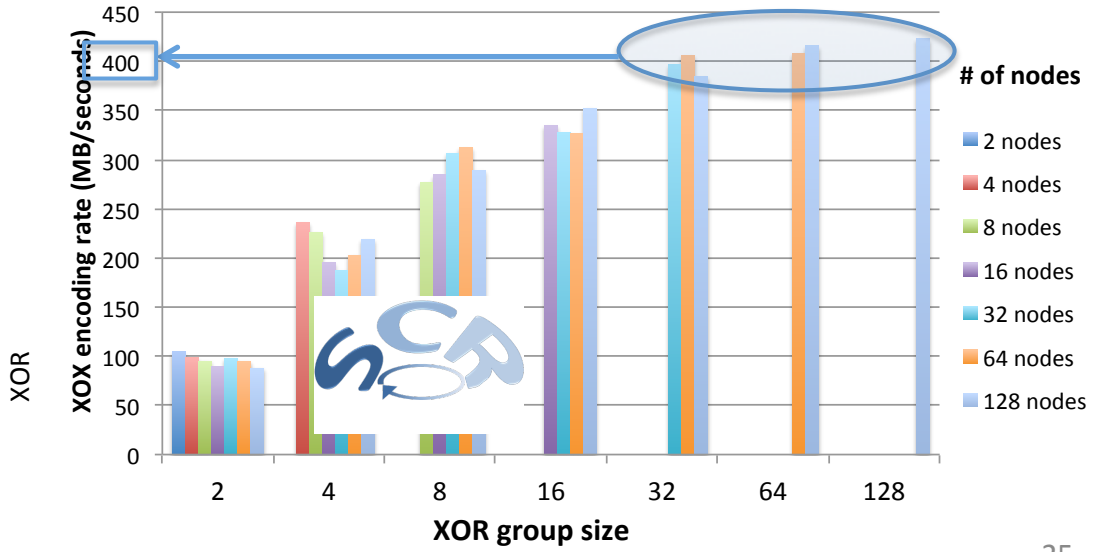
- **Checkpoint size per node**: 29GB
  - TSUBAME nodes memory: 58GB



Failure rates (failures/second) on TSUBAME2.0

Level-1

- **XOR throughput**: 400MB/s

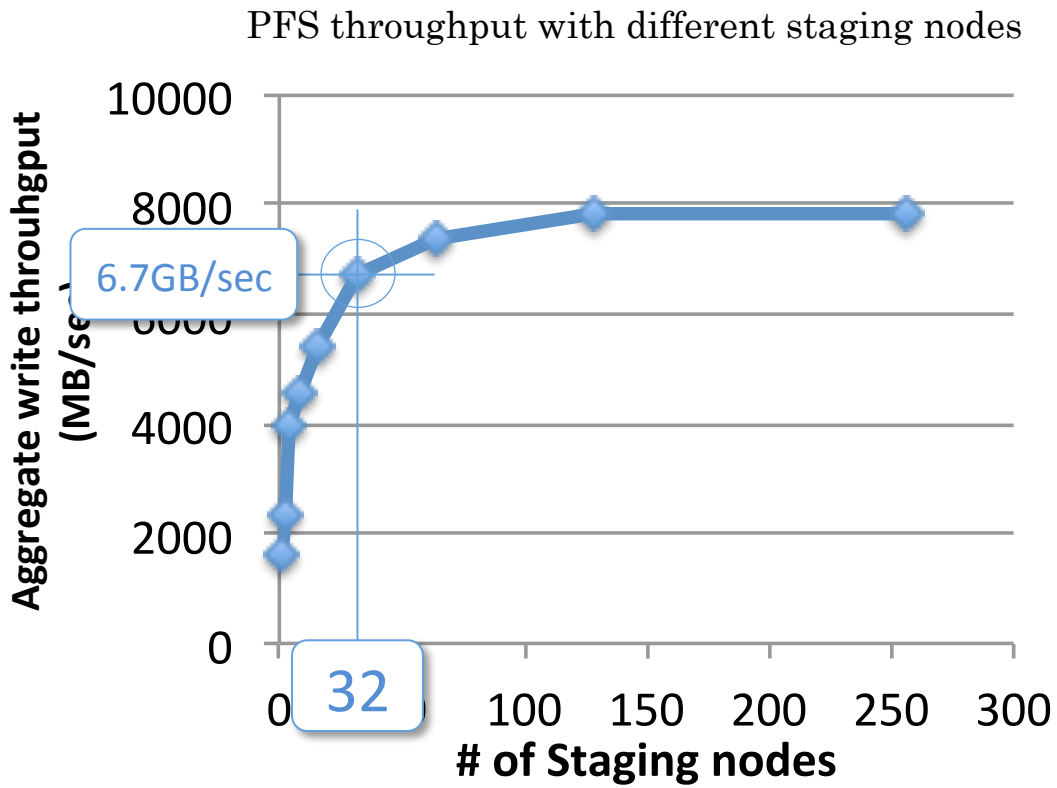


XOR encoding performance on TSUBAME2.0 using local SSDs

Staging node tuning for TSUBAME2.0

Level-2

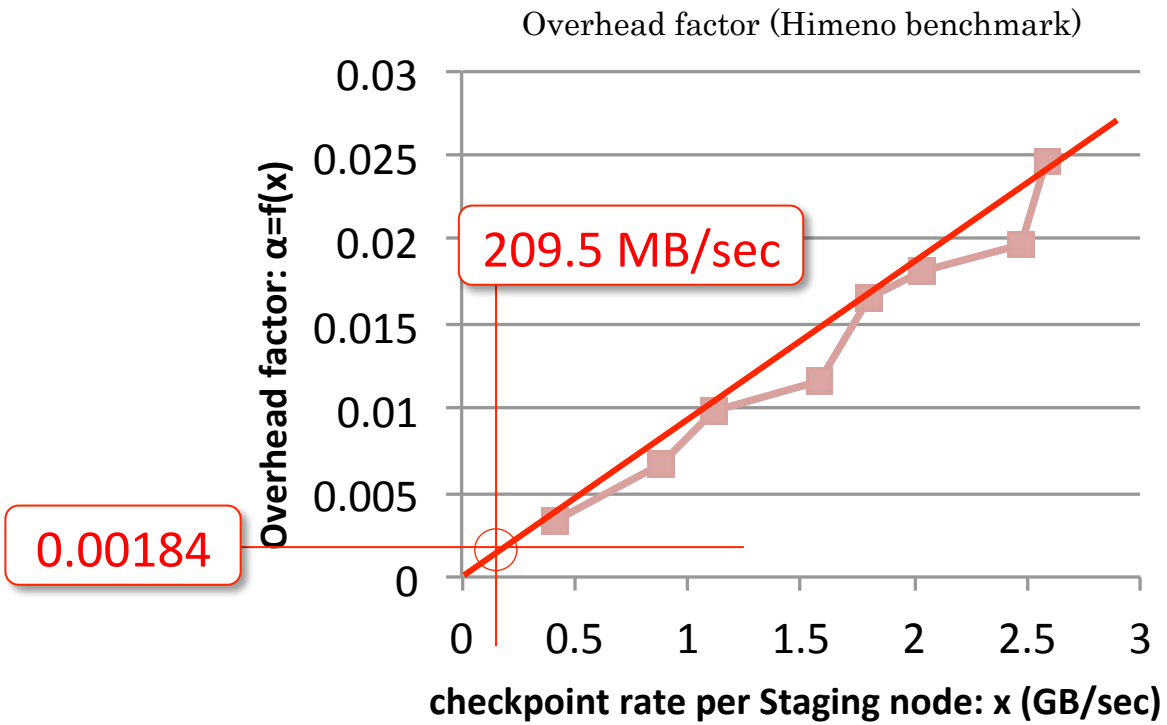
- **# of Staging nodes**: 32 nodes
  - 2.3% of TSUBAME2.0 thin nodes (1408 nodes)
- **PFS throughput**: 6.7GB/seconds
  - 209.5 MB/seconds* per Staging node
    * 6.7(GB/s) / 32(nodes) = 209.5



Overhead factor

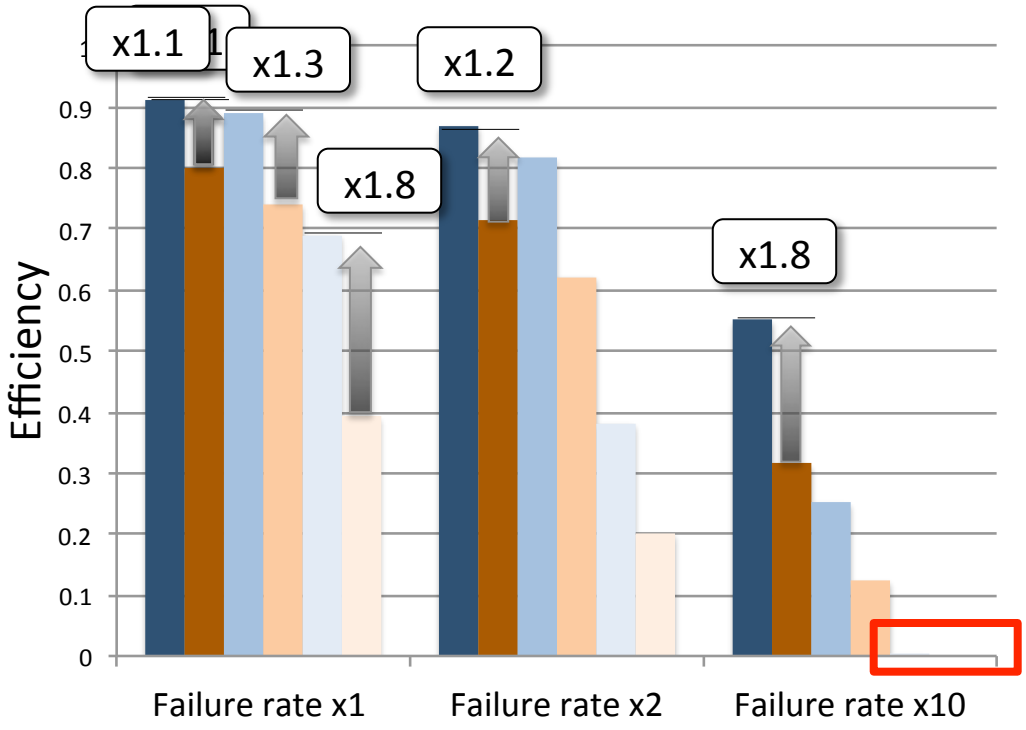
- **Overhead factor**: 0.00184 (0.184%)
  - For Himeno bechmark

RDMA ⇒ No CPU cycle, No redundant memcpy
RDMA read speed ⇒ 209.5MB/s < Network & Memory bandwidth



Efficiency: Non-blocking vs. blocking

The non-blocking method always achieves higher efficiency than the blocking method



$$Efficiency = \frac{ideal\ \ runtime}{expected\ \ runtime}$$

$ideal\ \ runtime$: No failure and No checkpoint

$expected\ \ runtime$: Computed by the models

- PFS cost x1 / Non-blocking
- PFS cost x1 / Blocking
- PFS cost x2 / Non-blocking
- PFS cost x2 / Blocking
- PFS cost x10 / Non-blocking
- PFS cost x10 / Blocking

One TSUBAME2.0 node MTBF: 2.57 years
# of Nodes: 1408 nodes

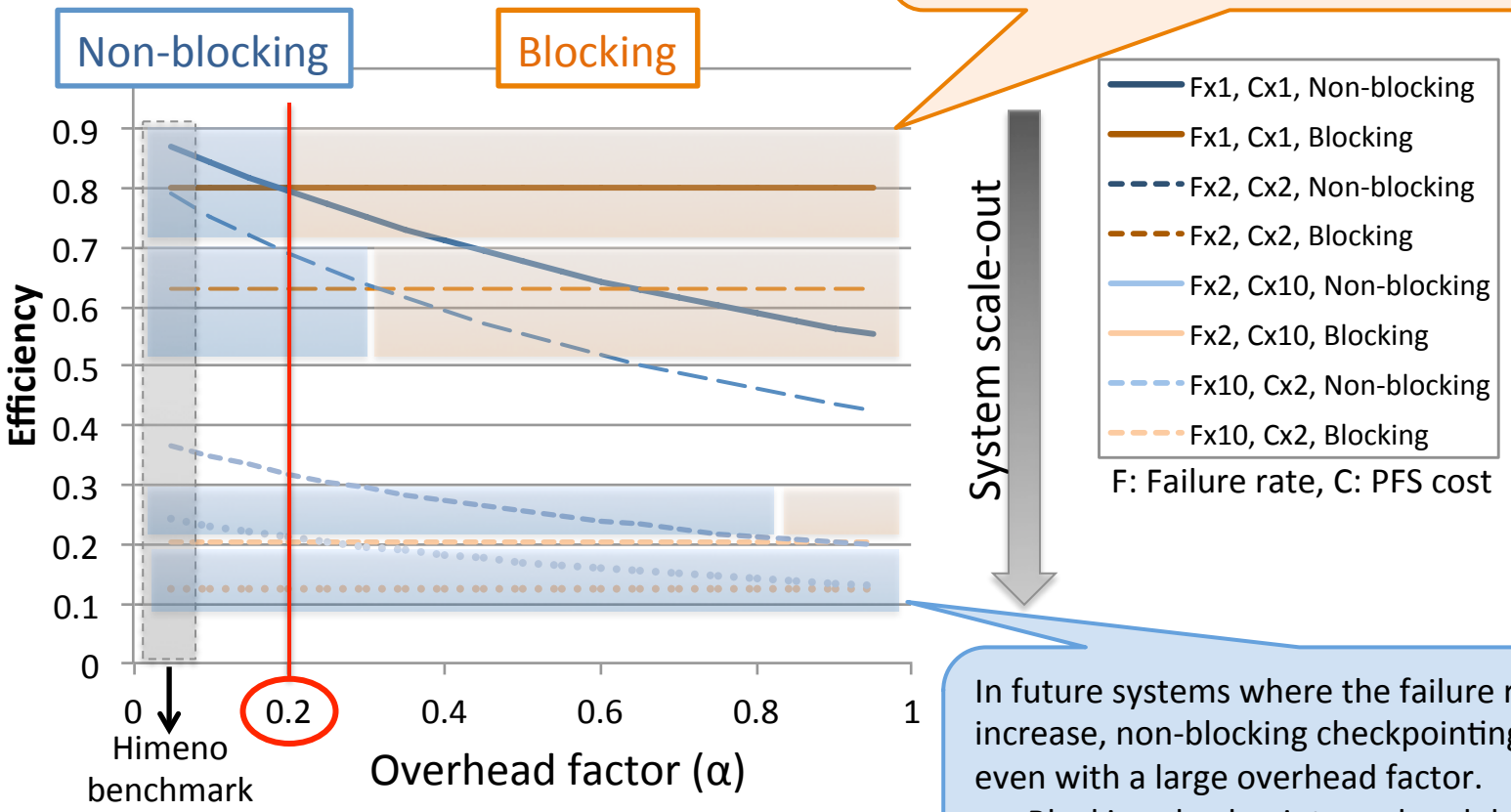
x10 scale-out →

No computation progresses !!

Overhead factor: Non-blocking vs. Blocking

Other applications case whose overhead factor becomes bigger

If overhead factor is over 0.2, blocking checkpointing can become more efficient in current system



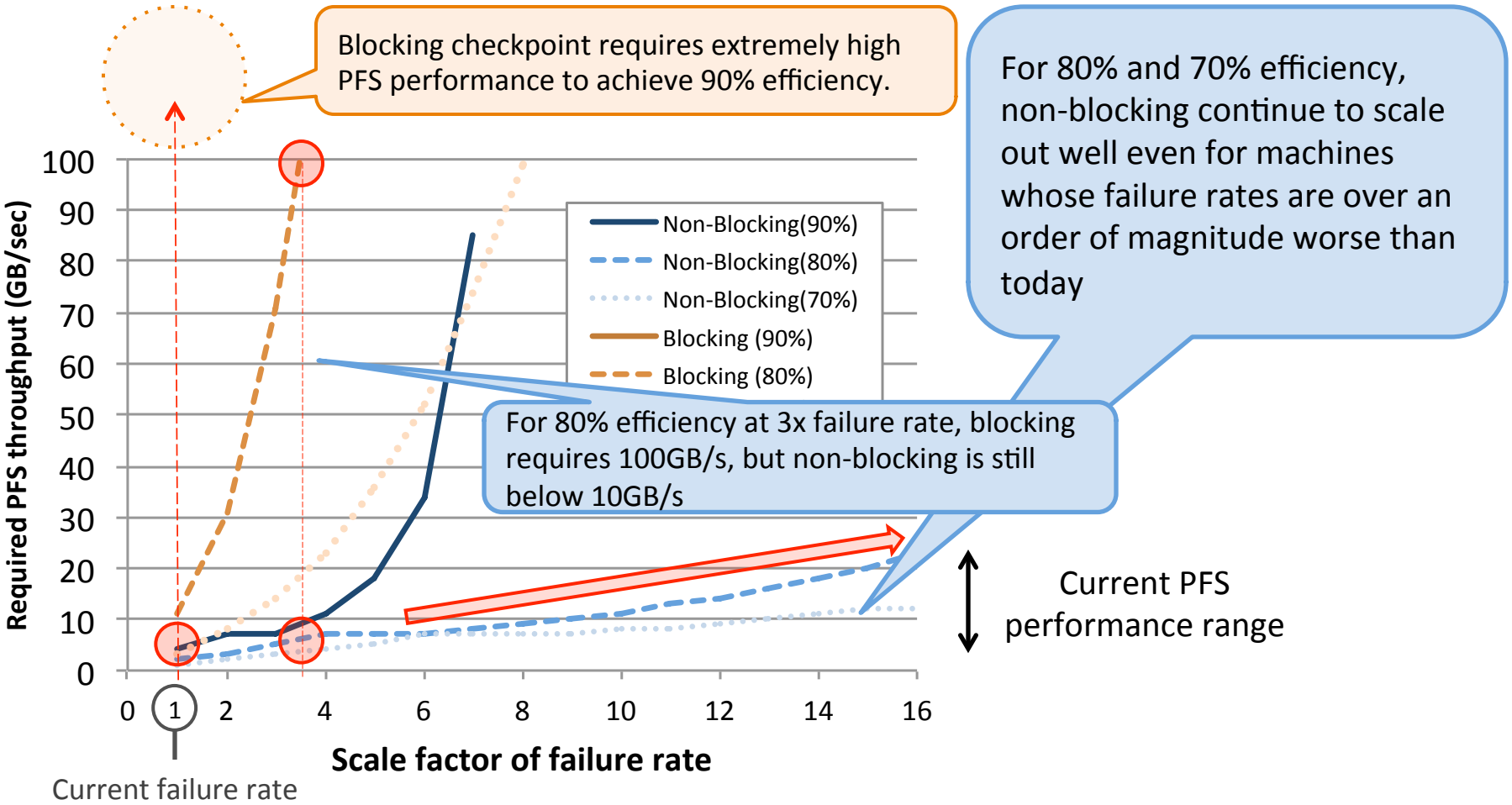
F: Failure rate, C: PFS cost

In future systems where the failure rates and cost increase, non-blocking checkpointing can be effective even with a large overhead factor.
=> Blocking checkpoint overhead dominate the runtime more than overhead factor by non-blocking

Required PFS performance to meet given application efficiency

When building a reliable data center or supercomputer, two major concerns are monetary cost of the PFS and the PFS throughput required to maintain high efficiency …

=> predict required PFS performance with the models



Conclusion

- Developed an non-blocking checkpointing system
  - Write checkpoint data in the background using RDMA
- Markov model of the non-blocking checkpointing
  - Optimal multi-level checkpoint interval
  - Non-blocking v.s. Blocking checkpoint
    - Higher efficiency (1.1 ~ 1.8x) on current and future systems
  - High efficiency (up to 80%) with low PFS throughput

Q & A

Speaker:

Kento Sato (佐藤 賢斗)
kent@matsulab.is.titech.ac.jp
Tokyo Institute of Technology (Tokyo Tech)
Research Fellow of the Japan Society for the Promotion of Science
http://matsu-www.is.titech.ac.jp/~kent/index_en.html

Co-authers

Adam Moody, Kathryn Mohror, Todd Gamblin, Bronis R de. Supinski,
Naoya Maruyama, Satoshi Matsuoka

Acknowledgement

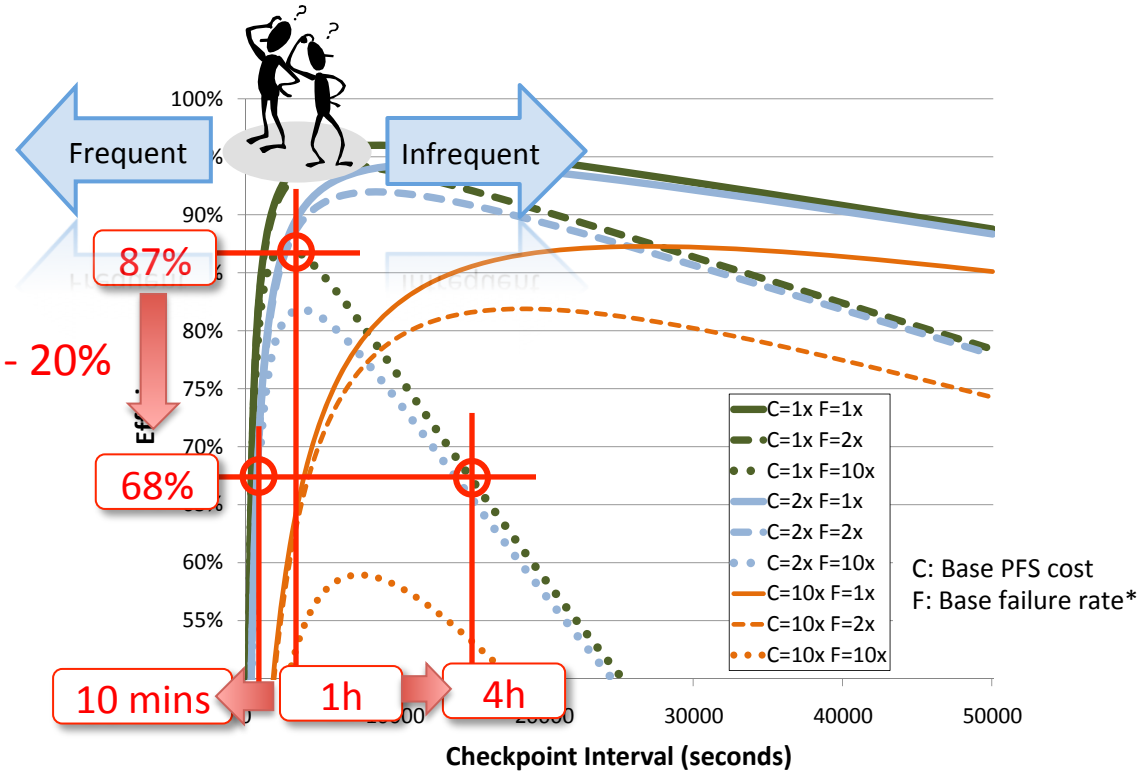
This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52- 07NA27344. LLNL-PRES-599833-DRAFT.This work was also supported by Grant-in-Aid for Research Fellow of the Japan Society for the Promotion of Science (JSPS Fellows) 24008253, and Grant-in-Aid for Scientific Research S 23220003.

BACKUP SLIDES

Background

Optimal checkpoint interval is significant

- Blindly determined interval causes low efficiency for an application's runtime with increasing failure rate



Blocking Multi-level checkpoint interval & efficiency

Blocking multi-level checkpointing

$$Efficiency = \frac{ideal\_runtime}{expected\_runtime}$$

$ideal\_runtime$: No failure and No checkpoint

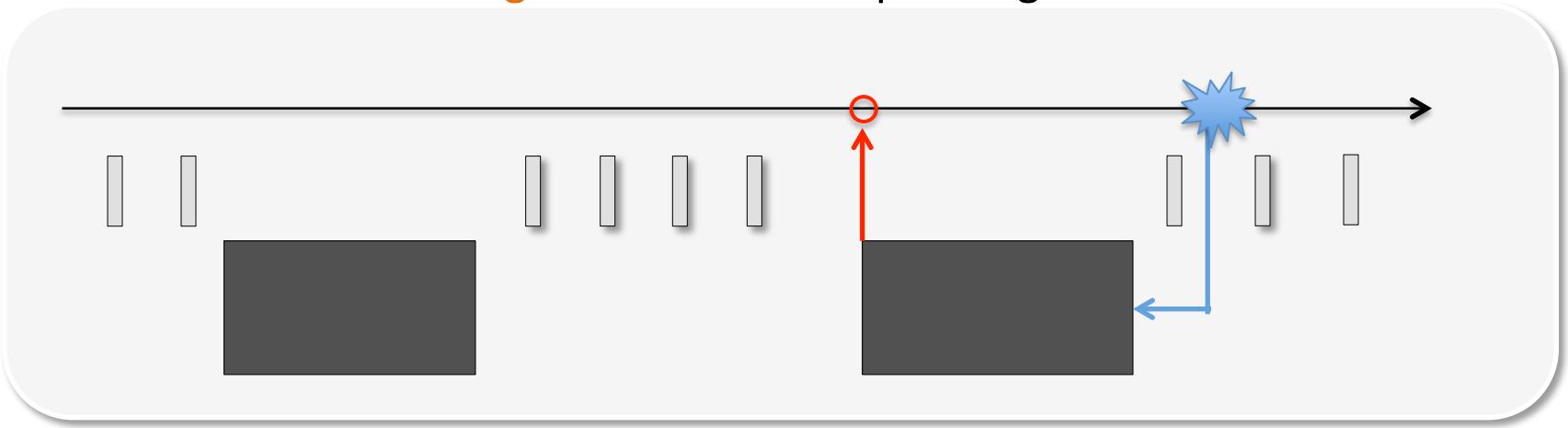
$expected\_runtime$: Computed by the blocking MLC model

Source: A. Moody, G. Bronevetsky, K. Mohror, and B. R. de Supinski, "Design, Modeling, and Evaluation of a Scalable Multi-level Checkpointing System," in Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC 10).

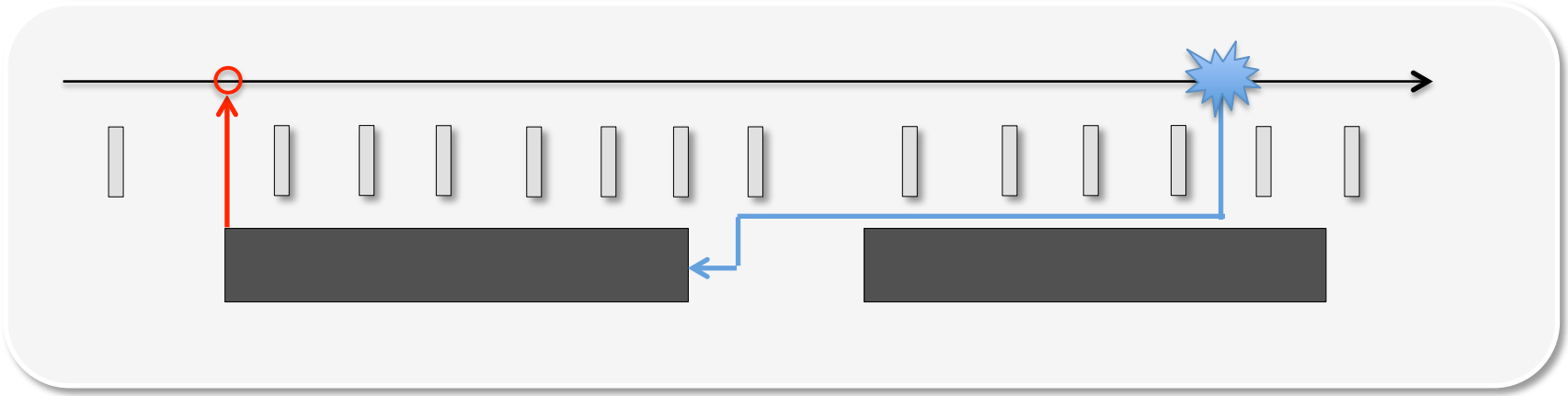
*LLNL clusters (Coastal, Hera, Atlas)

Optimal checkpoint interval is significant

Blocking multi-level checkpointing



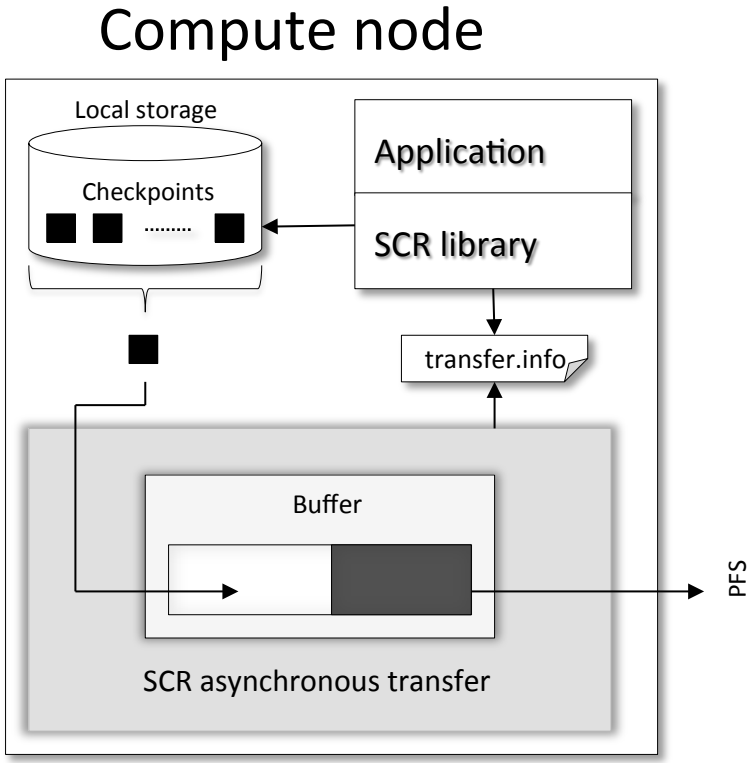
Non-blocking multi-level checkpointing



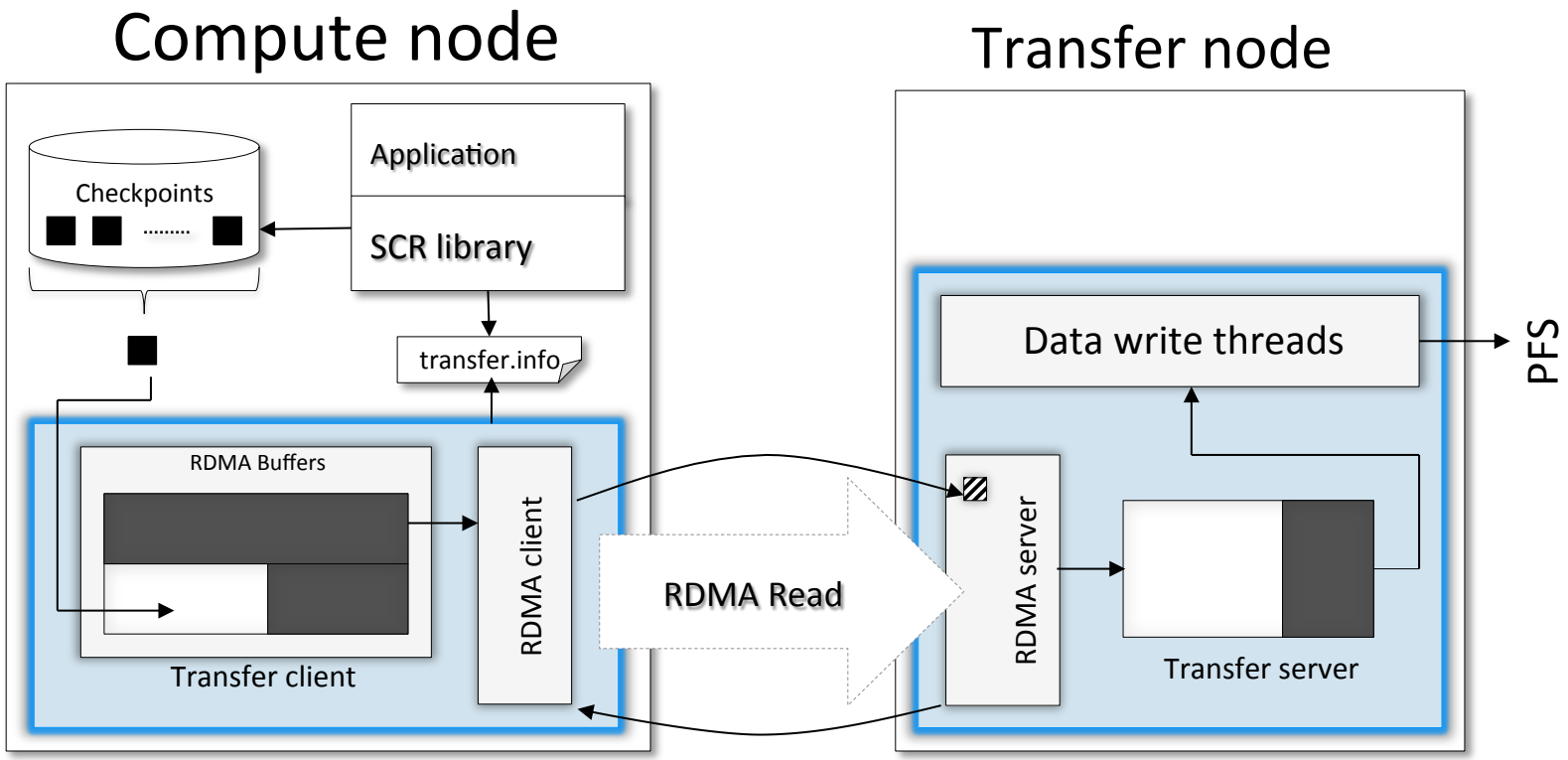
Non-blocking checkpointing implimentation

RDMA checkpoint transfers

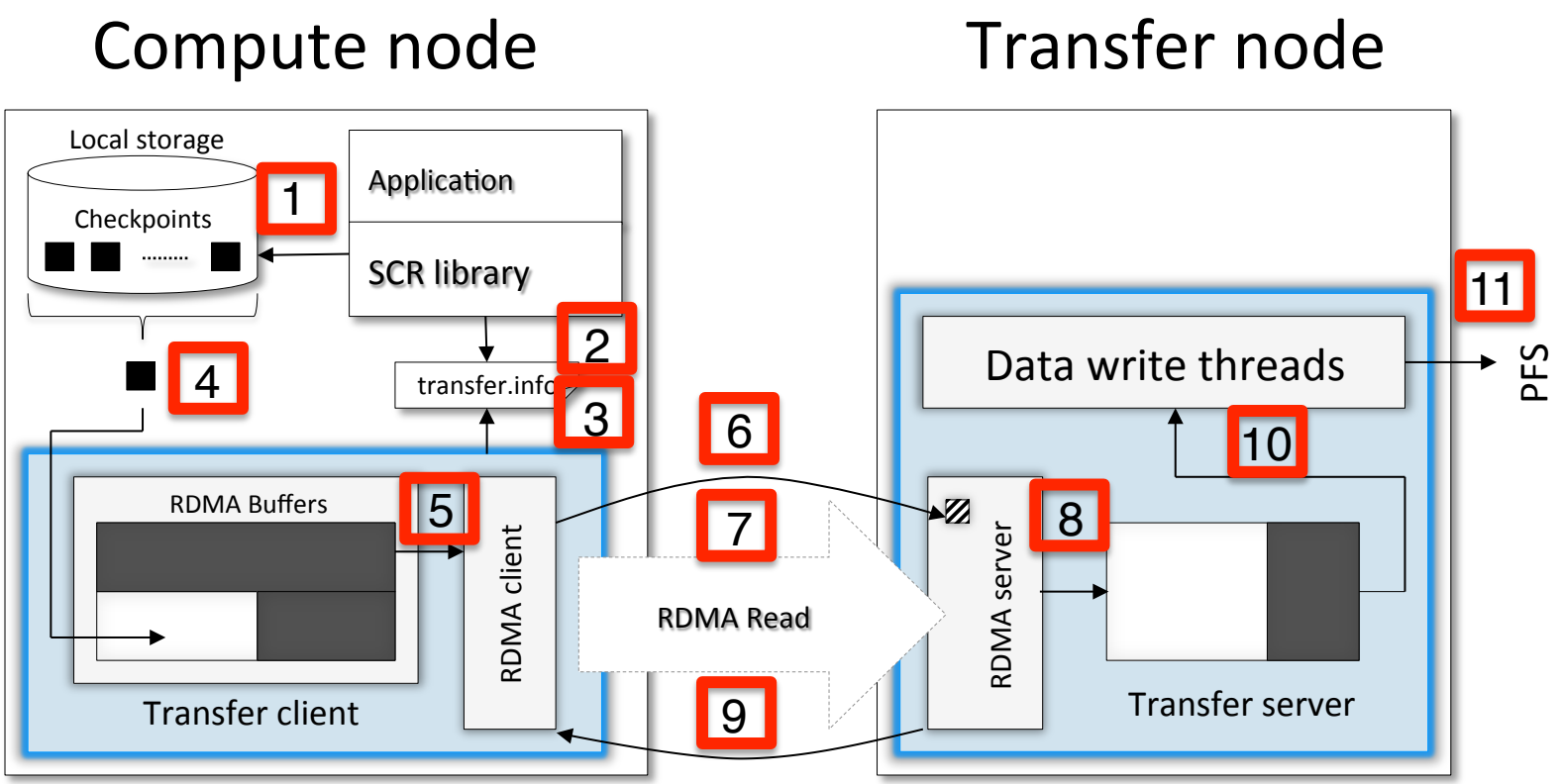
- Original asynchronous checkpointing in SCR



RDMA checkpoint transfers

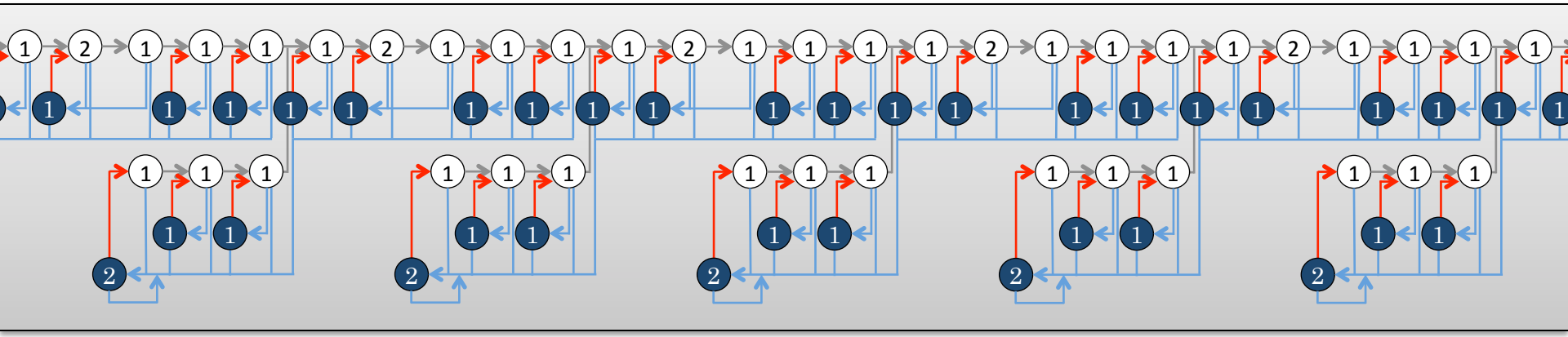


RDMA checkpoint transfers

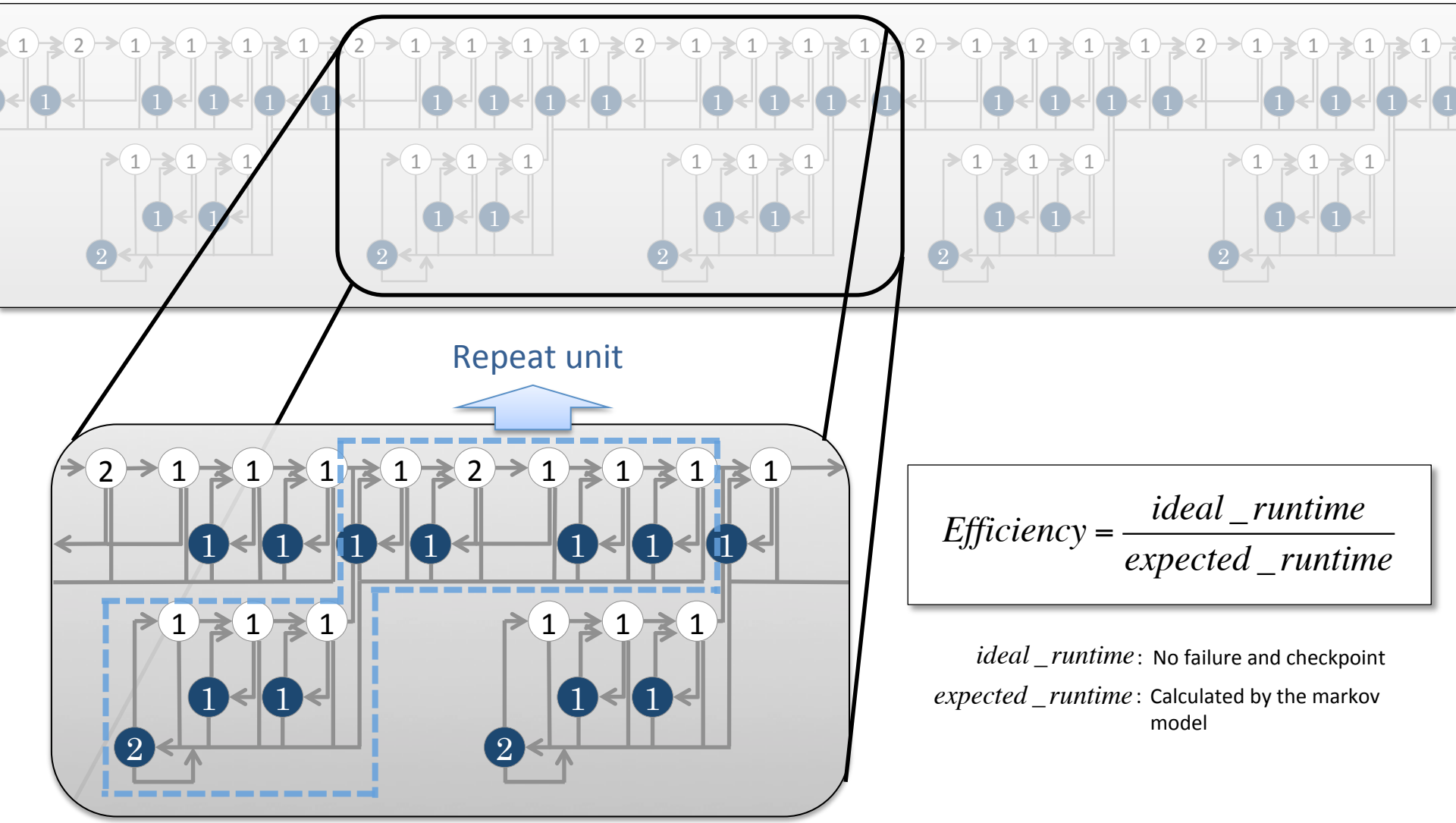


Modeling

Complete state transitions



Complete state transitions

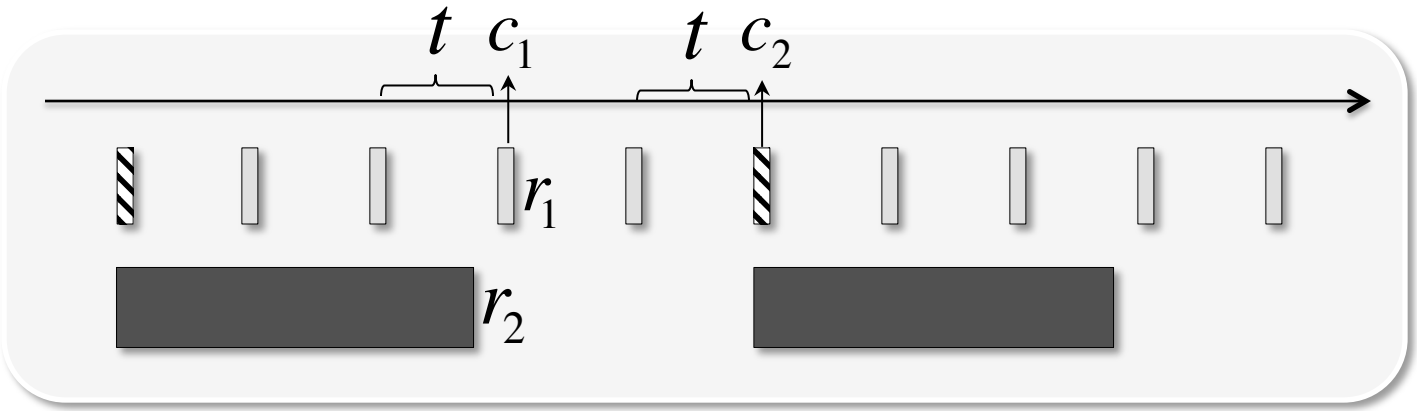


$$Efficiency = \frac{ideal\_runtime}{expected\_runtime}$$

$ideal\_runtime$: No failure and checkpoint

$expected\_runtime$: Calculated by the markov model

How to calculate *expected_runtime* ?



t : Interval

c_c : c-level checkpoint time

r_c : c-level recovery time

4 types of transitions

	Duration $t + c_k$	Duration r_k
No failure	k → $p_0(t+c_k)$ $t_0(t+c_k)$	k → $p_0(r_k)$ $t_0(r_k)$
Failure	k → i $p_i(t+c_k)$ $t_i(t+c_k)$	k → i $p_i(r_k)$ $t_i(r_k)$

$$p_0(T) = e^{-\lambda T}$$
$$t_0(T) = T$$
$$p_i(T) = \frac{\lambda_i}{\lambda}(1 - e^{-\lambda T})$$
$$t_i(T) = \frac{1 - (\lambda T + 1) \cdot e^{-\lambda T}}{\lambda \cdot (1 - e^{-\lambda T})}$$

λ_i : i-level checkpoint time

$$\lambda = \sum \lambda_i$$

$p_0(T)$: No failure for T seconds

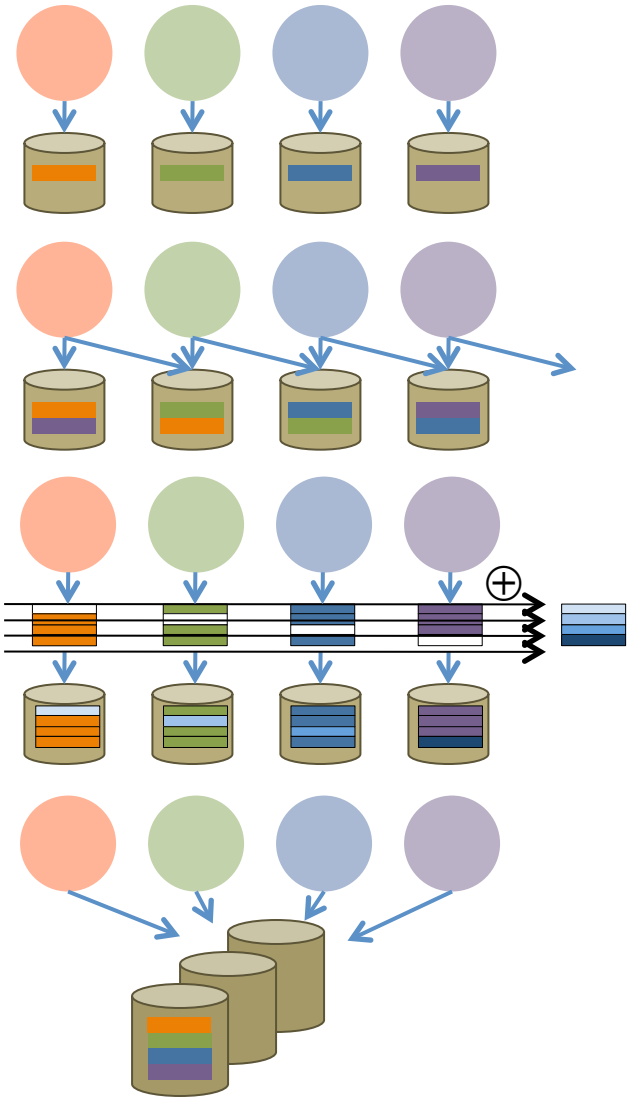
$t_0(T)$: Expected time when $p_0(T)$

$p_i(T)$: i - level failure for T seconds

$t_i(T)$: Expected time when $p_i(T)$

SCR: The Scalable Checkpoint/Restart Library

SCR uses multiple checkpoint levels for performance and resiliency

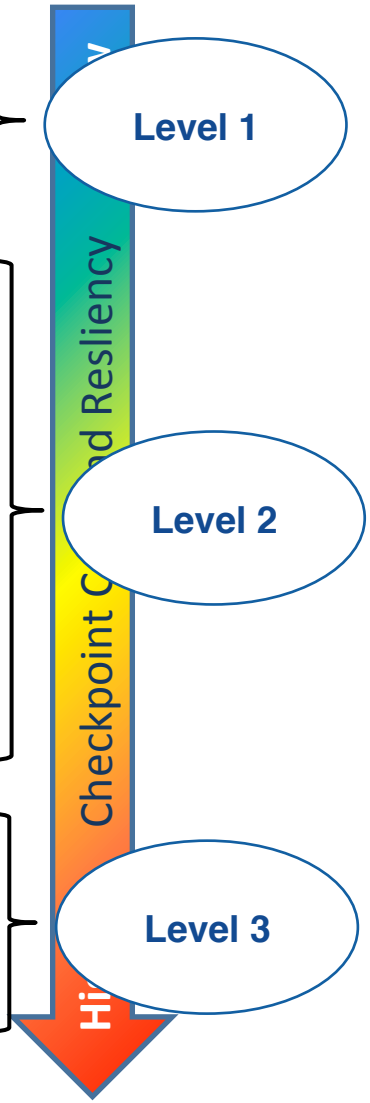


Local: Store checkpoint data on node's local storage, e.g. disk, memory

Partner: Write to local storage and on a partner node

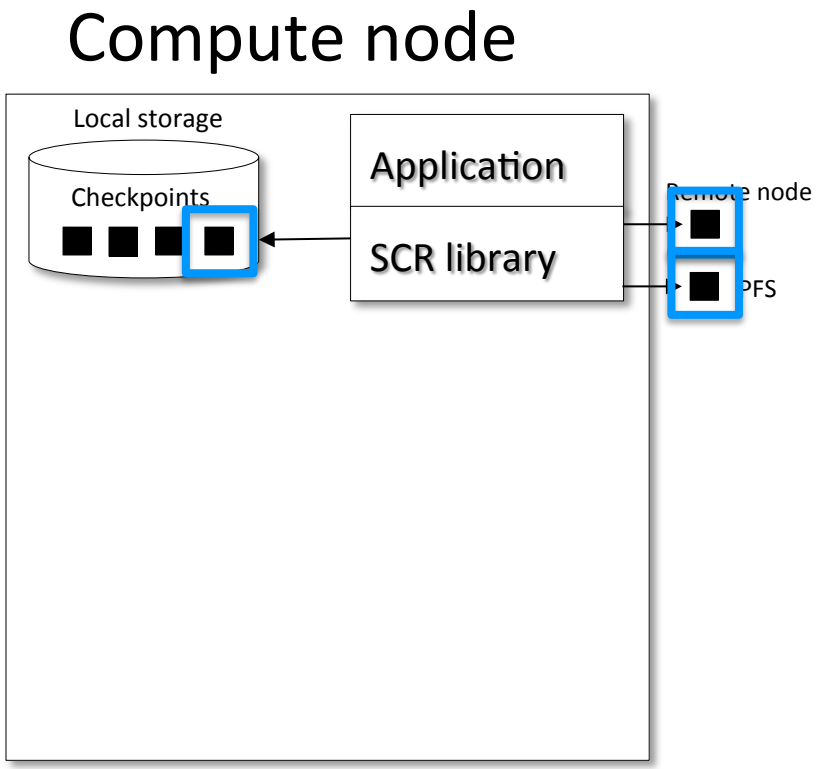
XOR: Write file to local storage and small sets of nodes collectively compute and store parity redundancy data (RAID-5)

Stable Storage: Write to parallel file system



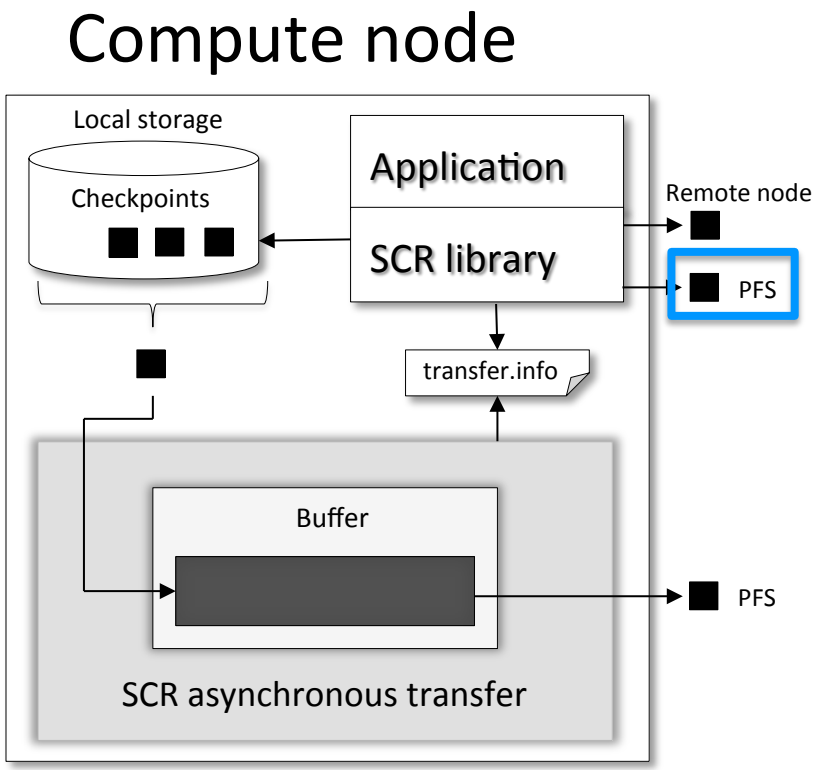
Checkpint/Restart with SCR

- SCR caches each checkpoint in local storage first
  - The user-defined number of the most recent checkpoints are cached
  - If the number of cached checkpoint exceeds the defined number, deletes older checkpoints
- SCR applies redundancy schemes and write the checkpoint to
  - Remote node-local storage with the specified redundancy schemes
    - e.g.) Partner, XOR
  - PFS (with specified frequency)
- On a failure, SCR tries to restart from a top level of the storage hierarchy



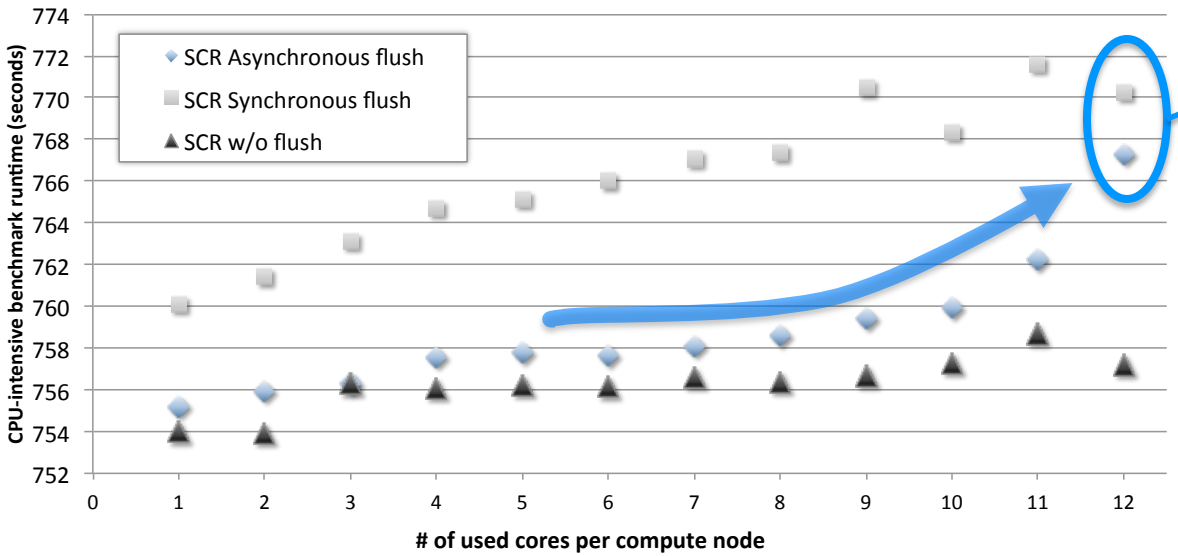
Flush by SCR

- SCR copies checkpoint to PFS with specified frequency (=> flush)
  - e.g.) If frequency is 10, SCR flushes every 10 checkpoints to PFS
- SCR supports two type of flushes
  - Synchronous flush
    - Blocks the application until the flush has completed
  - Asynchronous flush
    - Write transfer information as a file to request the flush
      - Immediately returns control to the application
    - Another process flushes the checkpoint to the PFS in the background



Flush performance

- Asynchronous from compute nodes checkpointing can affect an application runtime
  - An application uses especially all cores (12 cores) on nodes



Even asynchronous flush impacts on the program runtime as much as synchronous flush does because the background process use CPU time

- CPU-intensive program with no flush, sync/async flushes
  - CPU usage: over 90 %
- Sierra cluster at LLNL
  - checkpoint size: 200MB per process
  - Lustre (lscratchc, 30GB/s)

⇒ Motivated by the result, we developed an asynchronous checkpointing system using RDMA

Additional nodes for Stating

- Reduce Losing # of cores
  - 11cores (compute) + 1 core (checkpoint)

    => lose 8.3%
  - 1376nodes (compute) + 32 (checkpoint)

    ⇒lose 2.3%
- Compression without extra overhead on compute nodes

SCR works for codes that do globally-coordinated application-level checkpointing

```
int main(int argc, char* argv[]) {
  MPI_Init(argc, argv);

  for(int t = 0; t < TIMESTEPS; t+
   +)
  {
    /* ... Do work ... */

      checkpoint();
  }

  MPI_Finalize();
  return 0;
}
```

```
void checkpoint() {

  int rank;
  MPI_Comm_rank(MPI_COMM_WORLD,
   &rank);

  char file[256];
  sprintf(file, "rank_%d.ckpt",
   rank);

  FILE* fs = fopen(file, "w");
  if (fs != NULL) {
    fwrite(state, ..., fs);
    fclose(fs);
  }

  return;
}
```

SCR works for codes that do globally-coordinated application-level checkpointing

```
int main(int argc, char* argv[]) {
  MPI_Init(argc, argv);
  SCR_Init();

  for(int t = 0; t < TIMESTEPS; t+
   +)
  {
    /* ... Do work ... */

    int flag;
    SCR_Need_checkpoint(&flag);
    if (flag)
      checkpoint();
  }

  SCR_Finalize();
  MPI_Finalize();
  return 0;
}
```

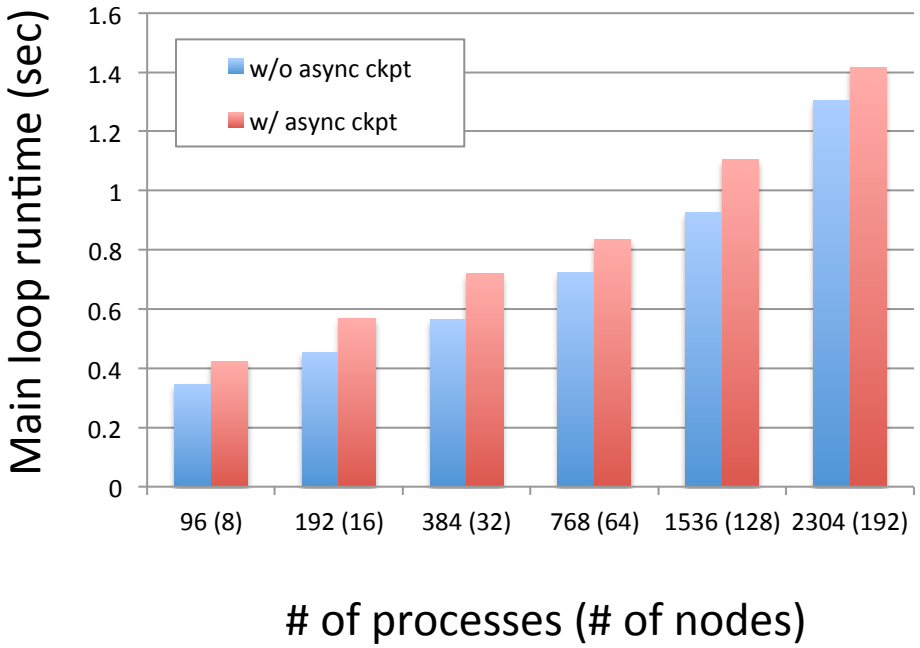
```
void checkpoint() {
  SCR_Start_checkpoint();

  int rank;
  MPI_Comm_rank(MPI_COMM_WORLD,
   &rank);

  char file[256];
  sprintf(file, "rank_%d.ckpt",
   rank);

  char scr_file[SCR_MAX_FILENAME];
  SCR_Route_file(file, scr_file);
  FILE* fs = fopen(scr_file, "w");
  if (fs != NULL) {
    fwrite(state, ..., fs);
    fclose(fs);
  }

  SCR_Complete_checkpoint(1);
  return;
}
```

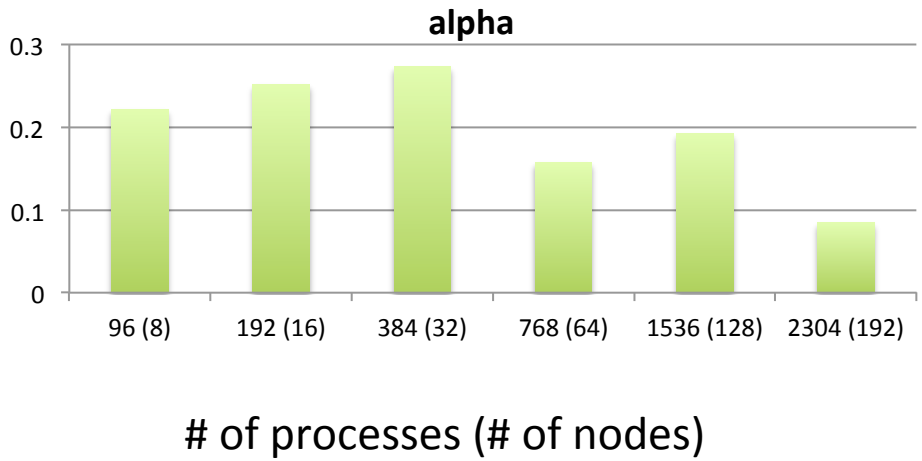
Evaluation

Noises with increasing # of procs

- Main Loop (x 10,000 loops)
  - ATS (10usec)
  - Allreduce(MAX)

* ATS (10usec) x 10,000 loops = 0.1 sec





Related work

Asynchronous I/O

- DataStager [Hasan et al. 2009]
  - Using additional nodes
  - As we observed, optimizing performance requires determination of the proper number of staging nodes for a given number of compute nodes
  - However, the comprehensive study on the problem is not shown nor do they present their solution.

H. Abbasi, M. Wolf, G. Eisenhauer, S. Klasky, K. Schwan, and F. Zheng, "DataStager: Scalable Data Staging Services for Petascale Applications," in Proceedings of the 18th ACM international symposium on High performance distributed computing, ser. HPDC '09. New York, NY, USA: ACM, 2009

Modeling of Checkpoint/Restart

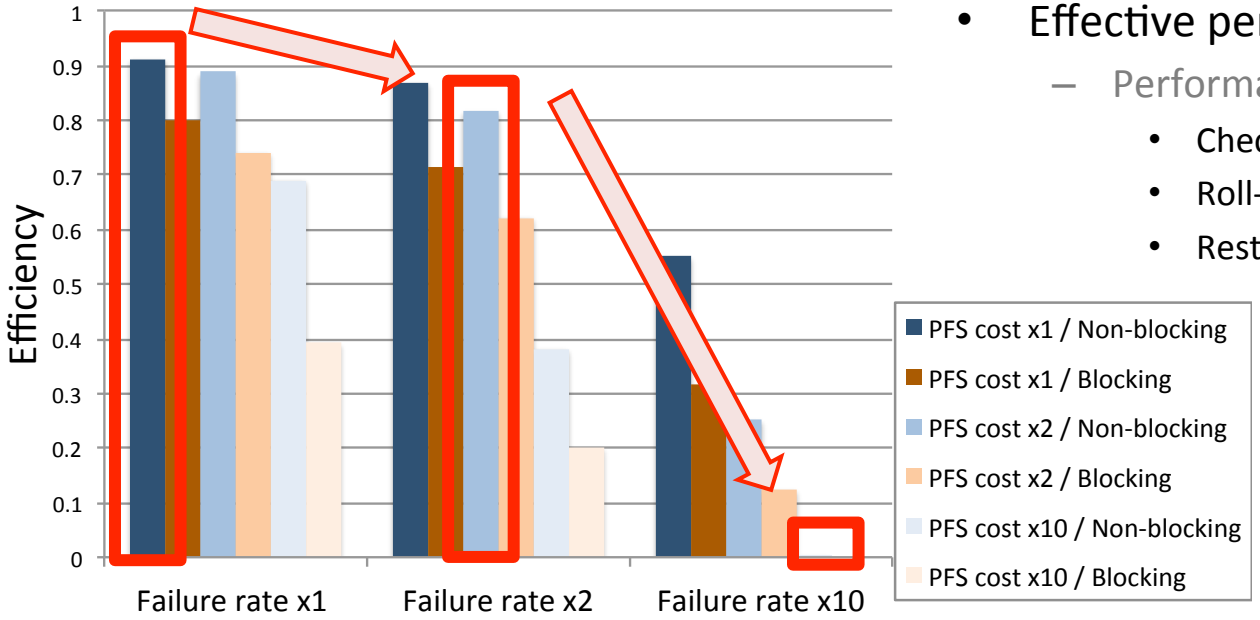
- Single-level model [Young et al. 1974]
- [Vaidya, 1994-1995]
  - Two-level
  - Single-level + forked checkpoints
  - Two-level + forked checkpoints

- Our model
  - Arbitrary N-level checkpoint
  - Level-N checkpoint can overlap with lower level checkpoints

BACKUP SLIDES

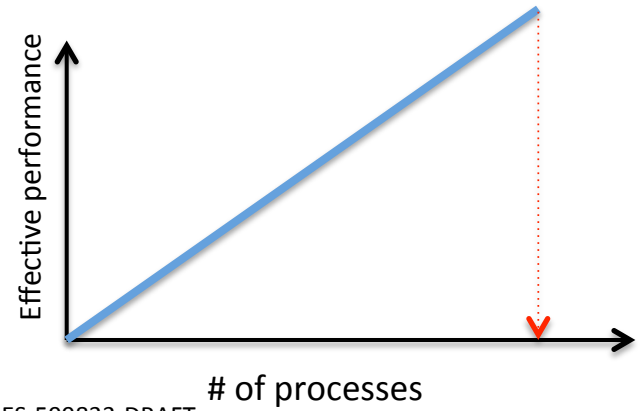
FOR Design

Scaling problem on long-running apps

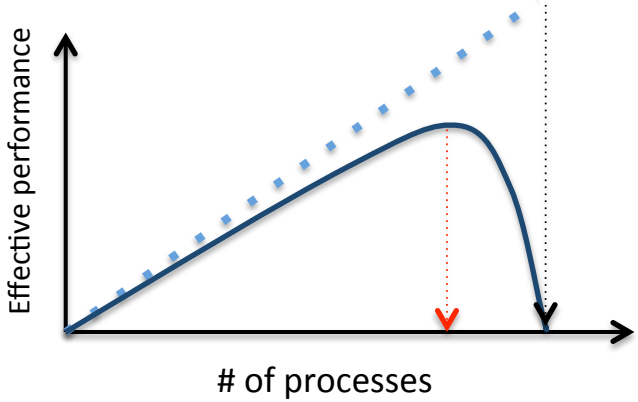


- Effective performance (EP):
  - Performance including
    - Checkpoint
    - Roll-back
    - Restart

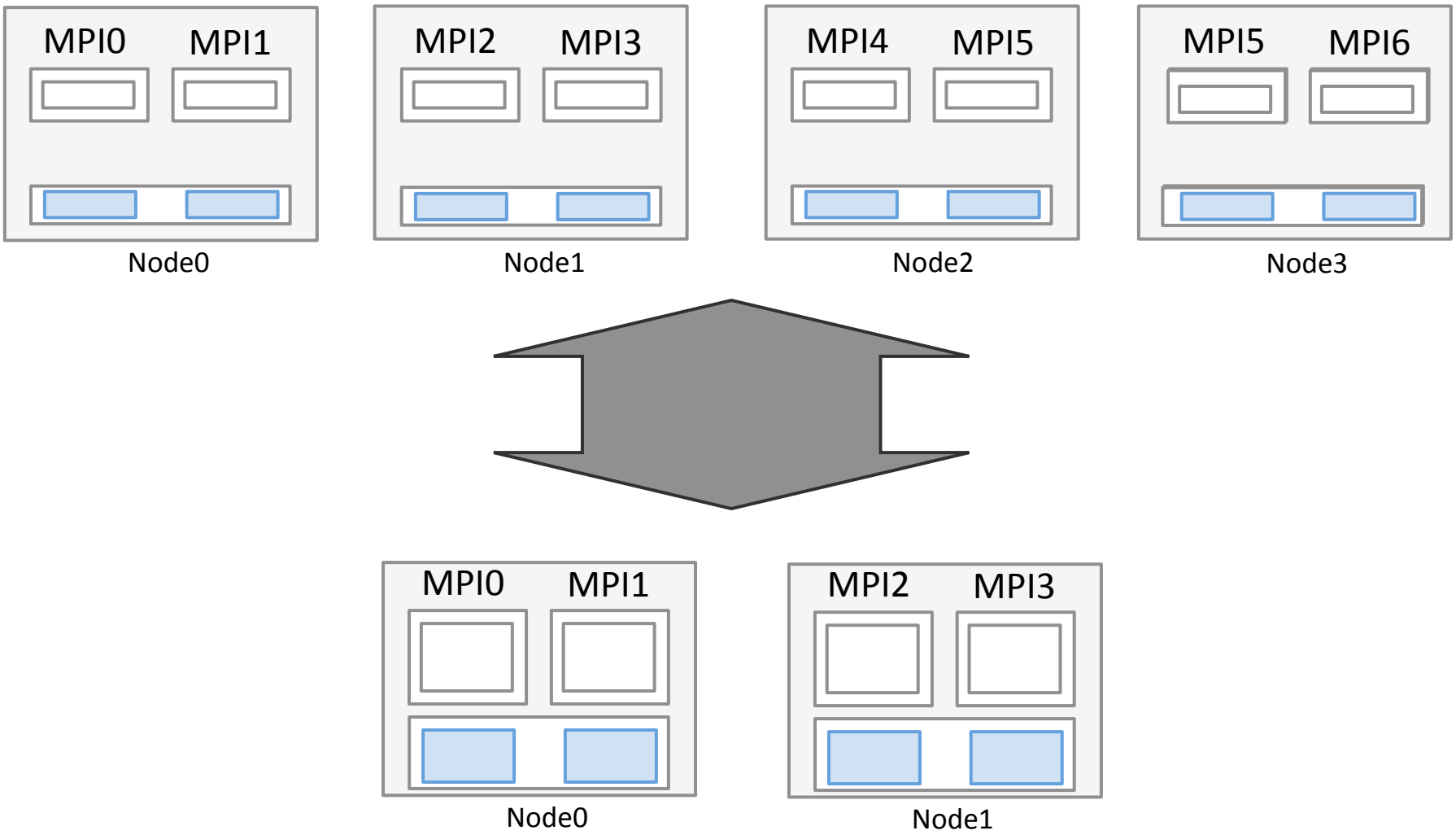
EF ≈ Performance



EF = Performance x Efficiency

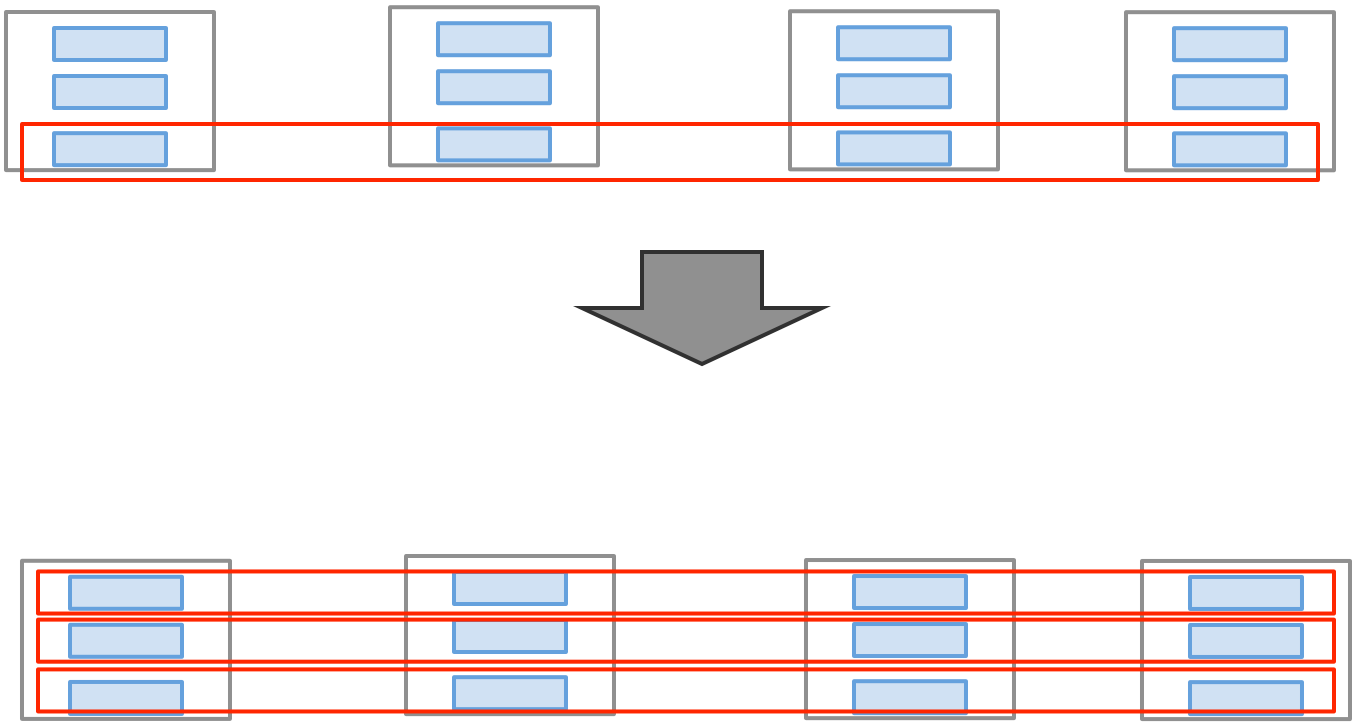


Dynamic scale out/down



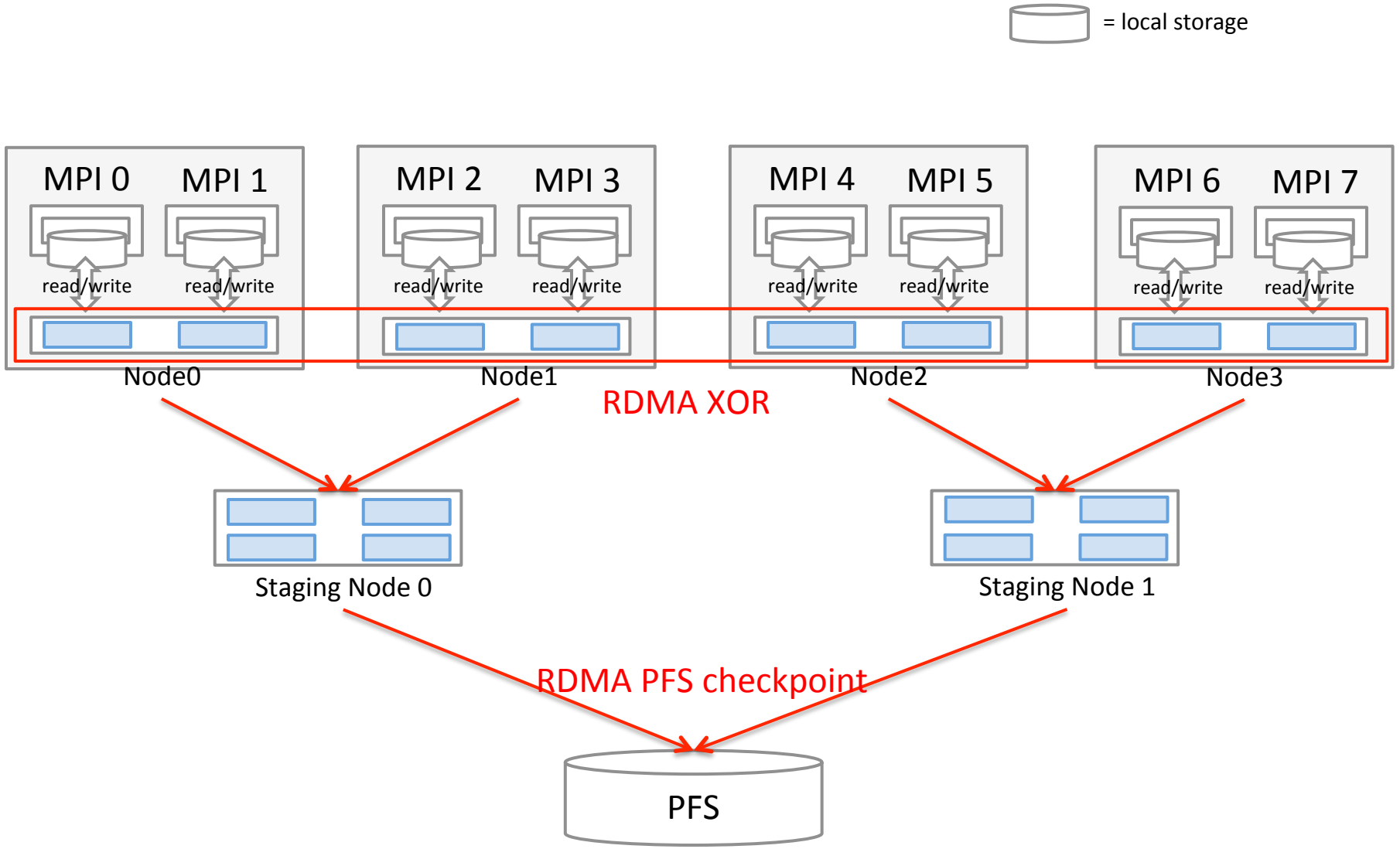
XOR Performance

XOR in Parallel



Integration

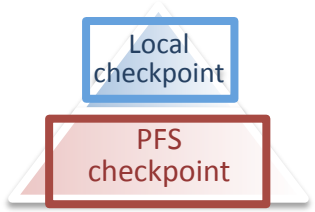
CRUISE integration



Memo

FT Communication library

- Potential overhead
  - Detect failure
  - exclude slow nodes
  - failure rate varies
    - ノードがアサインされるまでわからない
  - アプリのリーソースの使用料によってfailureが違う
    - GPU or CPU
  - 時間によっても変わるから

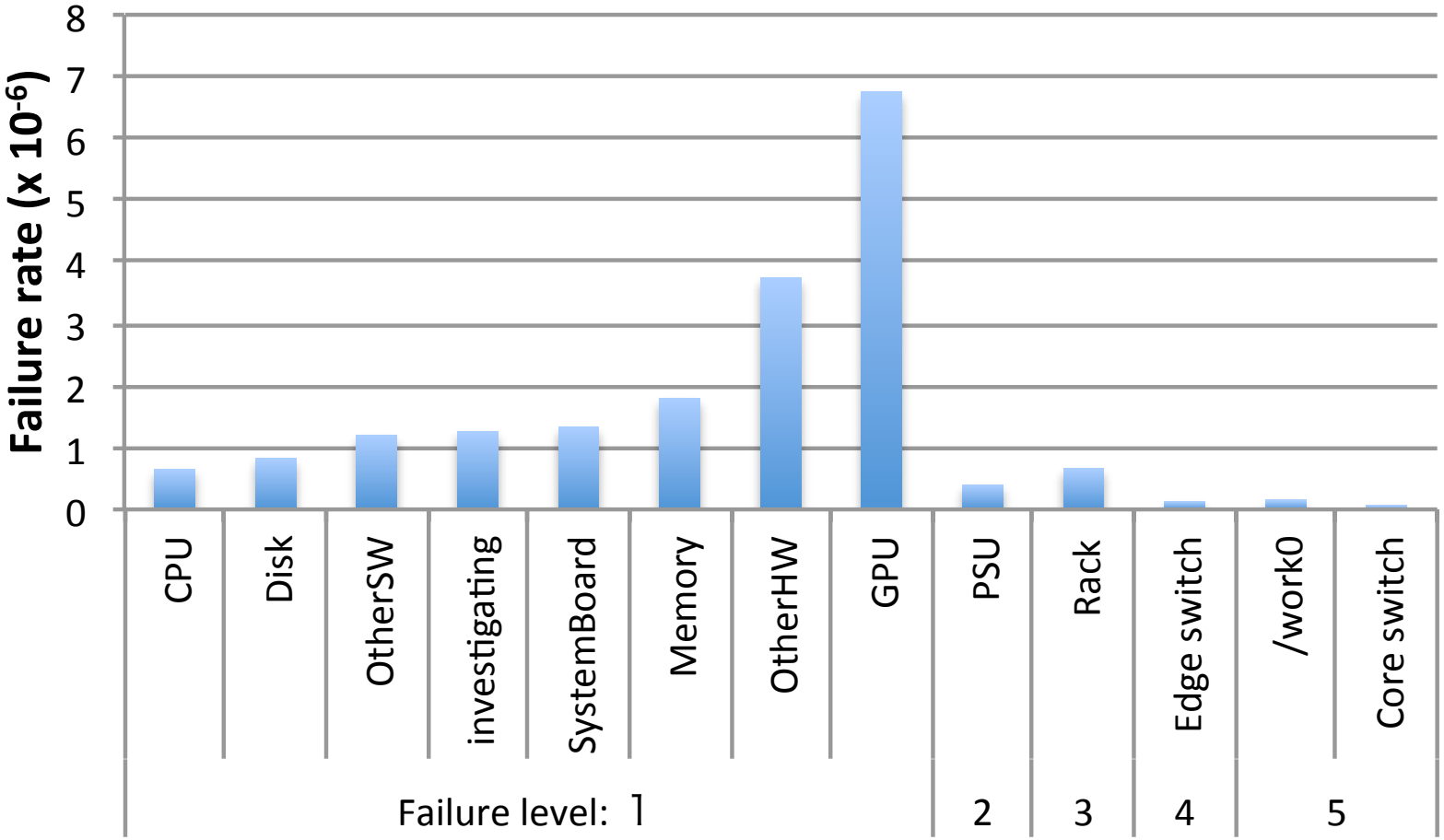


- We are Forcusing PFS checkpoint

Memo

- Background
  1. slow XOR
  2. 10x < 1x
  3. unstable performance
  4. failure trends 80:20 rule
  5. complicated model to use
  6. Potential overhead
     - exclude nodes
     - job submision
  7. 縮退運転
  8. 二重計算なども入れたい
  9. 峠を過ぎたらノードを増やしたい
- Proposal:
  - (1) In-memory asynchronous XOR checkpoint
  - (2, 7) Scaling
  - Reliable communication library
    - (3) Detect failure and slow nodes
    - (6, 7) Elastic MPI
  - FTE: fault tolerant library
    - (2) Optimal number of nodes
    - (4, 5) tuning the interval

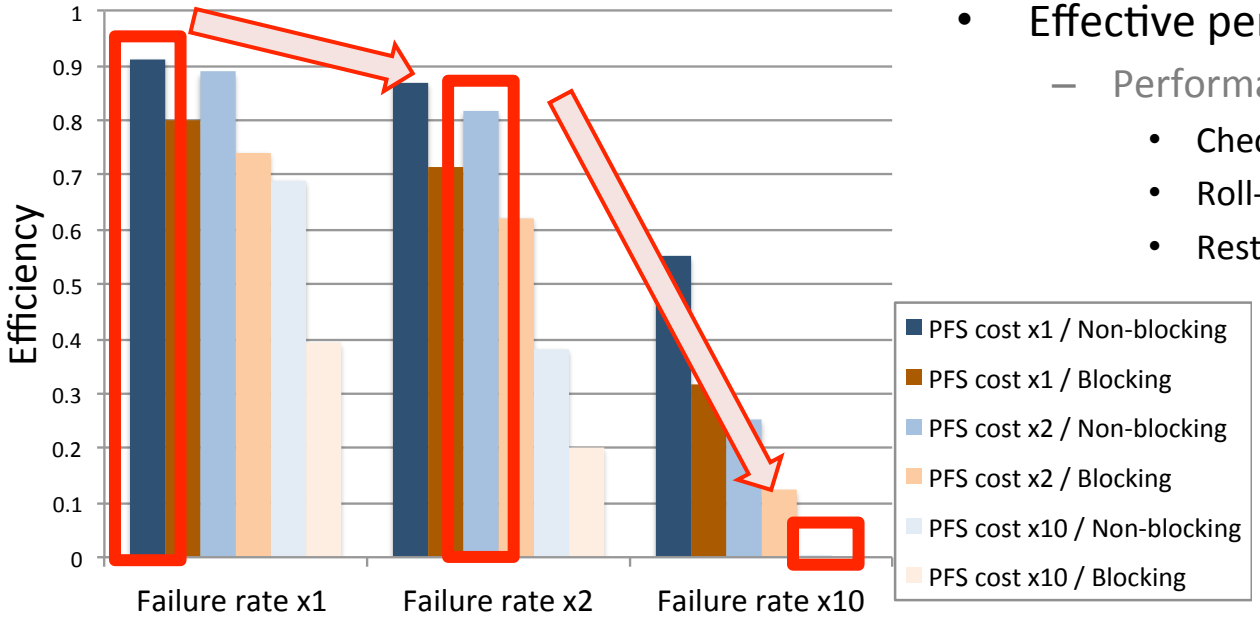
BACKUP SLIDES


Failure rate (x 10-6)
8
7
6
5
4
3
2
1
0
CPU
Disk
OtherSW
investigating
SystemBoard
Memory
OtherHW
GPU
PSU
Rack
Edge switch
/work0
Core switch
Failure level: 1
2
3
4
5

Future work & Next steps

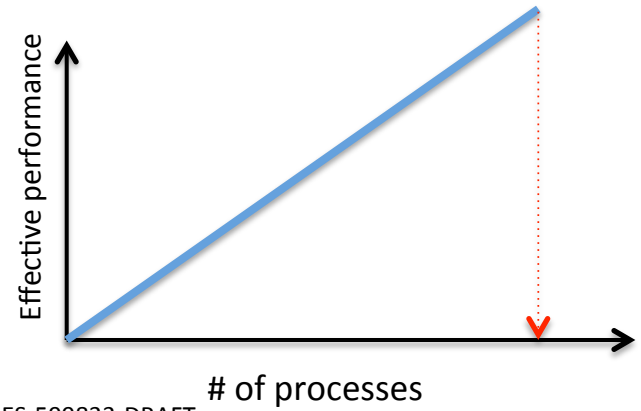
- Achieve high efficiency with a non-blocking PFS checkpoint, but …
  1. XOR checkpoint overhead
     - Frequent XOR checkpoints can also dominate a runtime in large failure rate even with high PFS throughput
  2. Scaling problem on long-running applications
     - Frequent roll-back/restart by large failure rate

Scaling problem on long-running apps

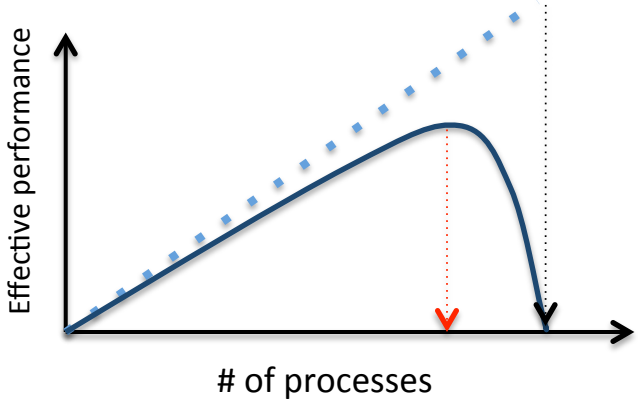


- Effective performance (EP):
  - Performance including
    - Checkpoint
    - Roll-back
    - Restart

EF ≈ Performance



EF = Performance x Efficiency

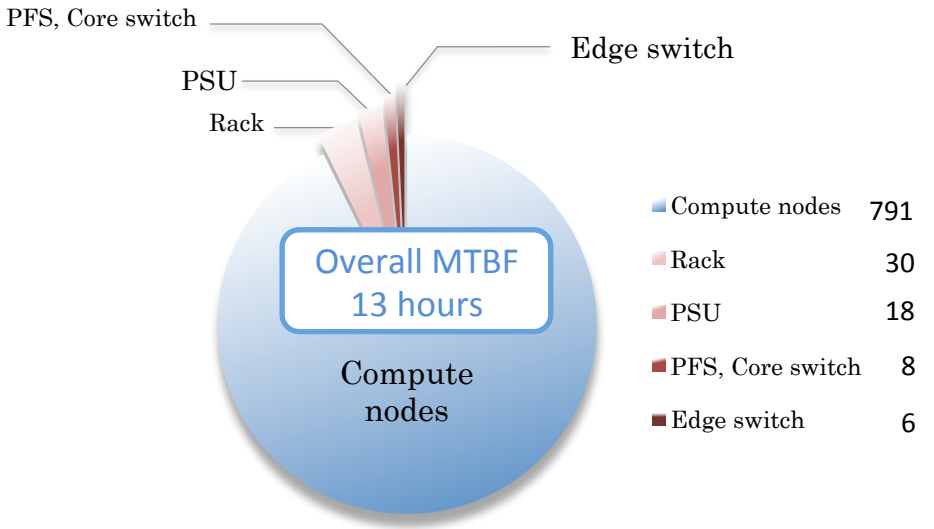


Future work & Next steps

- Achieve high efficiency with a non-blocking PFS checkpoint, but …
  1. XOR checkpoint overhead
     - Frequent XOR checkpoints can also dominate a runtime in large failure rate even with high PFS throughput
  2. Scaling problem on long-running applications
     - Frequent roll-back/restart by large failure rate
  3. Failure rates is not identically distributed
     - Optimal Interval cannot be determined statistically

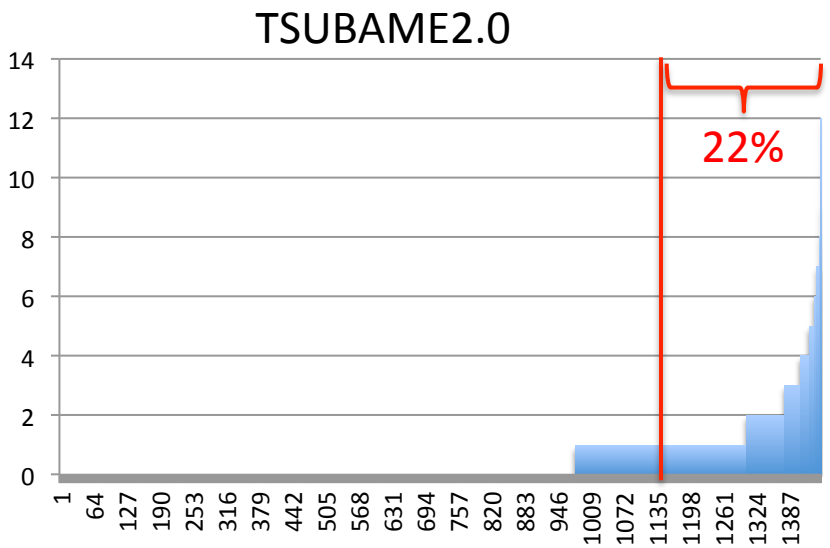
Non-uniform failure rate

- In future systems, MTBF is expected to decrease, but …
- Most failures occurs on small set of nodes
  - e.g.) TSUBAME2.0: 80% of failures on 22% of nodes
  - e.g.) TSUBAME1: 80% of failures on 39% of nodes



Failure analysis on TSUBAME2.0

Period: 1.5 years (Nov 1st, 2010 ~ April 6th 2012)
Observations: 962 node failures in total



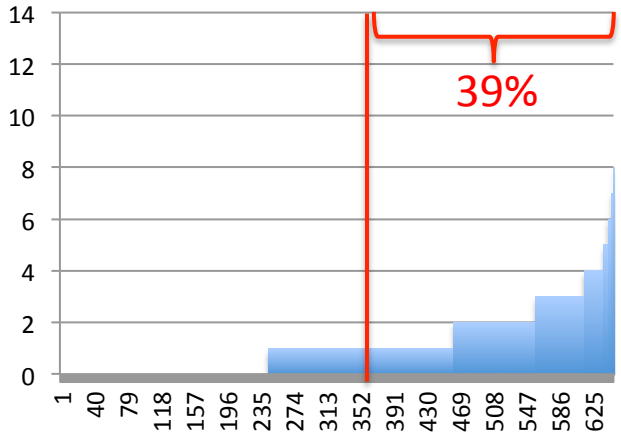
- System failure rate decrease as failure happens
  - Faulty nodes are excluded earlier than the other nodes

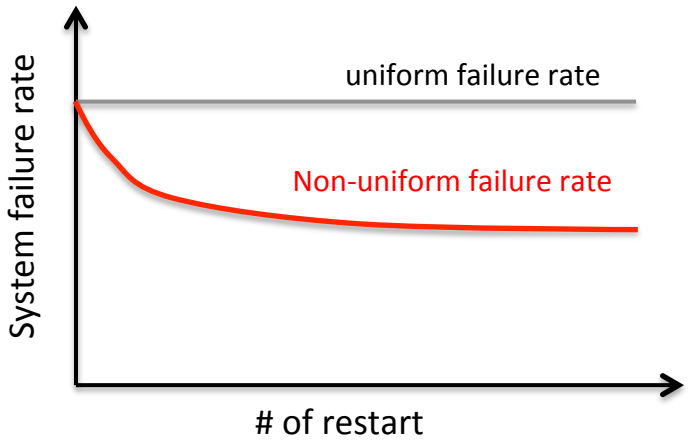
Non-uniform failure rate

- In future systems, MTBF is expected to decrease, but …
- Most failures occurs on small set of nodes
  - e.g.) TSUBAME2.0: 80% of failures on 22% of nodes
  - e.g.) TSUBAME1: 80% of failures on 39% of nodes

Failure analysis on TSUBAME1 (47.38 TFlops, 7th top500 in Nov 2006)

Period: 3.5 years (Oct 26th 2006 ~ March 21st, 2010)
Observations: 744 node failures in total





- System failure rate decrease as failure happens
  - Faulty nodes are excluded earlier than the other nodes

Future work & Next steps

- Achieve high efficiency with a non-blocking PFS checkpoint, but …
  1. XOR checkpoint overhead
     - Frequent XOR checkpoints can also dominate a runtime in large failure rate even with high PFS throughput
  2. Scaling problem on long-running applications
     - Frequent roll-back/restart by large failure rate
  3. Non-uniform failure rates
     - Optimal Interval cannot be determined statistically

Towards Asynchronous and Adaptive Multi-level Checkpoint/Restart

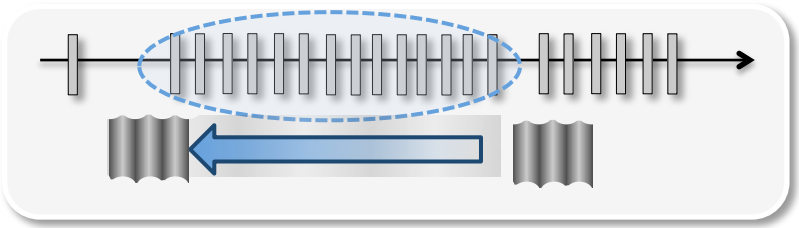
Kento Sato[†1,2], Adam Moody[†3], Kathryn Mohror[†3], Todd Gamblin[†3],
Bronis R. de Supinski[†3], Naoya Maruyama[†4,5] and Satoshi Matsuoka[†1,5,6,7]

†1 Tokyo Institute of Technology
†2 Research Fellow of the Japan Society for the Promotion of Science
†3 Lawrence Livermore National Laboratory
†4 RIKEN Advanced institute for Computational Science
†5 Global Scientific information and Computing Center
†6 National Institute of Informatics
†7 JST/CREST

LLNL-PRES-599833-DRAFT

Remaining issues on MLC

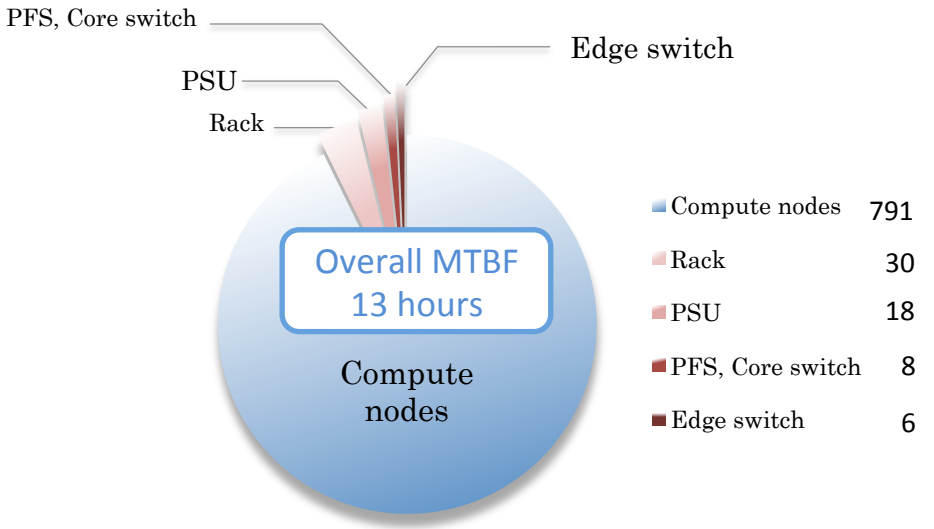
- Achieve high efficiency with a non-blocking PFS checkpoint, but …
  1. XOR checkpoint overhead
     - Frequent XOR checkpoints can also dominate a runtime in large failure rate even with high PFS throughput

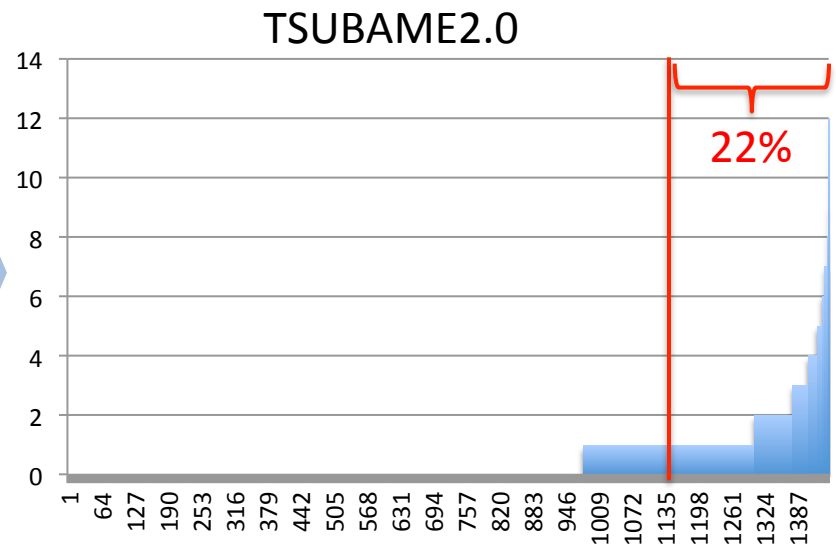


  2. Non-uniform failure rates
     - Optimal Interval cannot be determined statistically

Non-uniform failure rate

- In future systems, MTBF is expected to decrease, but …
- Most failures occurs on small set of nodes
  - e.g.) TSUBAME2.0: 80% of failures on 22% of nodes
  - e.g.) TSUBAME1: 80% of failures on 39% of nodes





Failure analysis on TSUBAME2.0

Period: 1.5 years (Nov 1st, 2010 ~ April 6th 2012)
Observations: 962 node failures in total

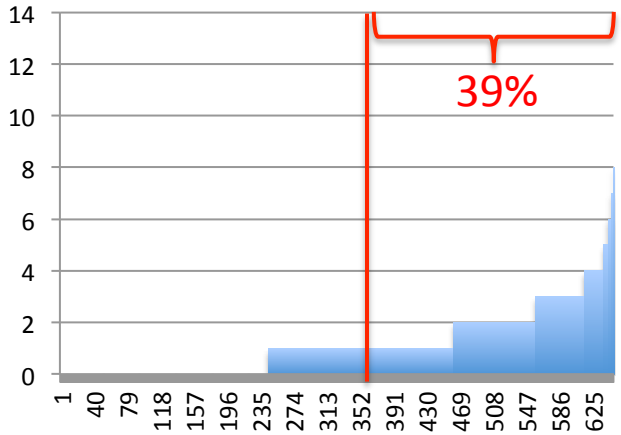
- System failure rate decrease as failure happens
  - Faulty nodes are excluded earlier than the other nodes

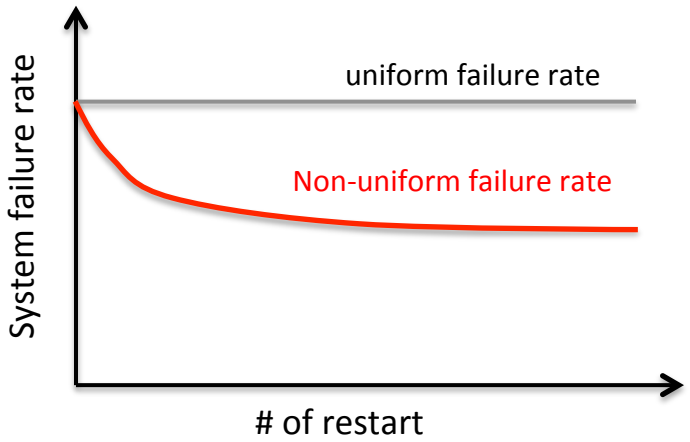
Non-uniform failure rate

- In future systems, MTBF is expected to decrease, but …
- Most failures occurs on small set of nodes
  - e.g.) TSUBAME2.0: 80% of failures on 22% of nodes
  - e.g.) TSUBAME1: 80% of failures on 39% of nodes

Failure analysis on TSUBAME1 (47.38 TFlops, 7th top500 in Nov 2006)

Period: 3.5 years (Oct 26th 2006 ~ March 21st, 2010)
Observations: 744 node failures in total



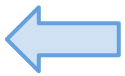


- System failure rate decrease as failure happens
  - Faulty nodes are excluded earlier than the other nodes

Objective, Proposal and Contributions

- **Objective**: More efficient MLC
  - Reduce XOR checkpoint
  - Adjust optimal checkpoint interval

- **Proposal:** Asynchronous and Adaptive MLC
  - Asynchronous XOR checkpoint using RDMA
  - Adaptive checkpoint interval on a restart

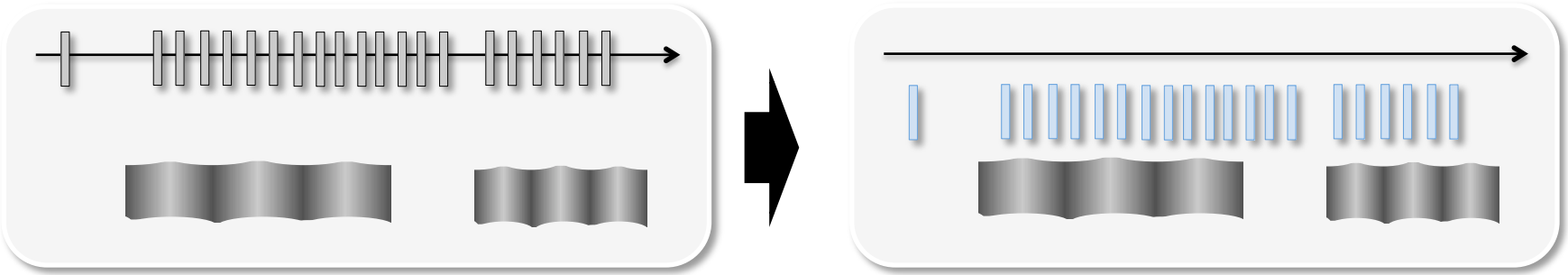


Focus on the internship

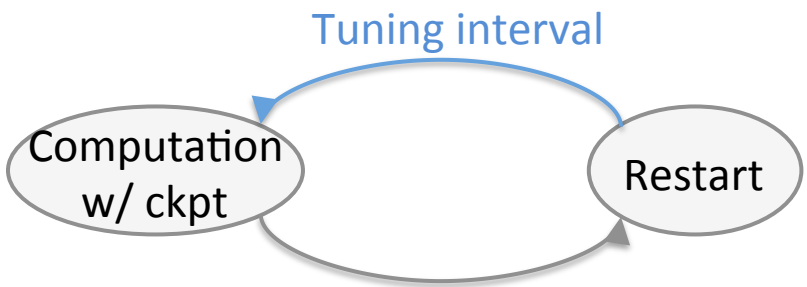
- **Contributions**:
  - Developed fault tolerant massaging library (FMI)
    - Achieve high throughput as MPI with large message size
  - Achieved Scalable XOR checkpointing with the number of nodes

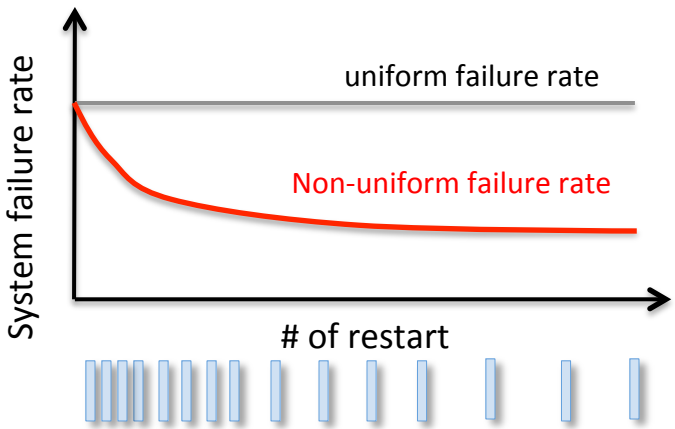
Asynchronous and Adaptive MLC

- Asynchronous XOR
  - Create XOR checkpoint using RDMA in the background

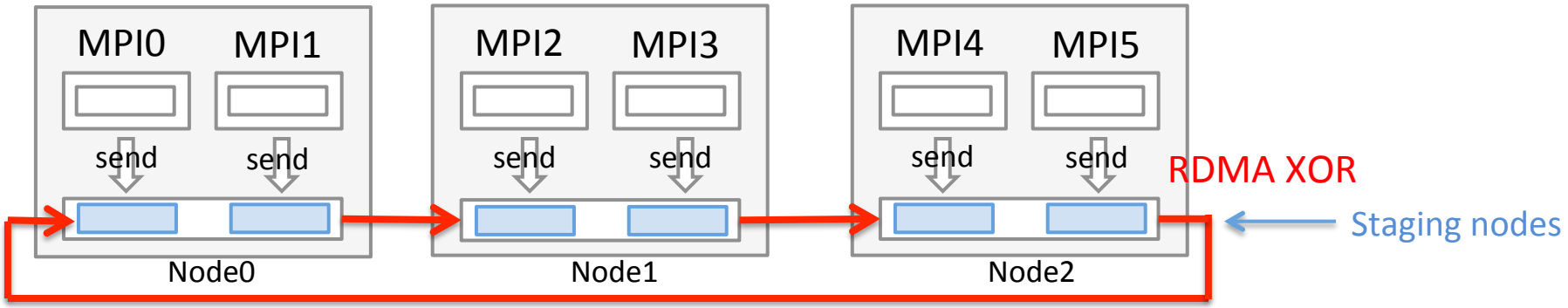


- Adaptive checkpoint interval



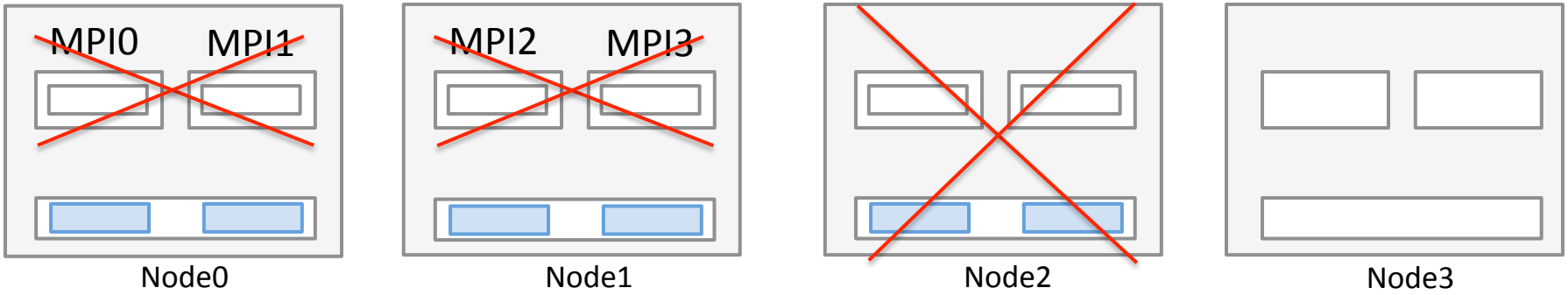


System overview



1. On a checkpoint, send to a staging client, then…
   - Staging nodes start XOR encoding in the background
   - Running processes can continue to run the computation

System overview



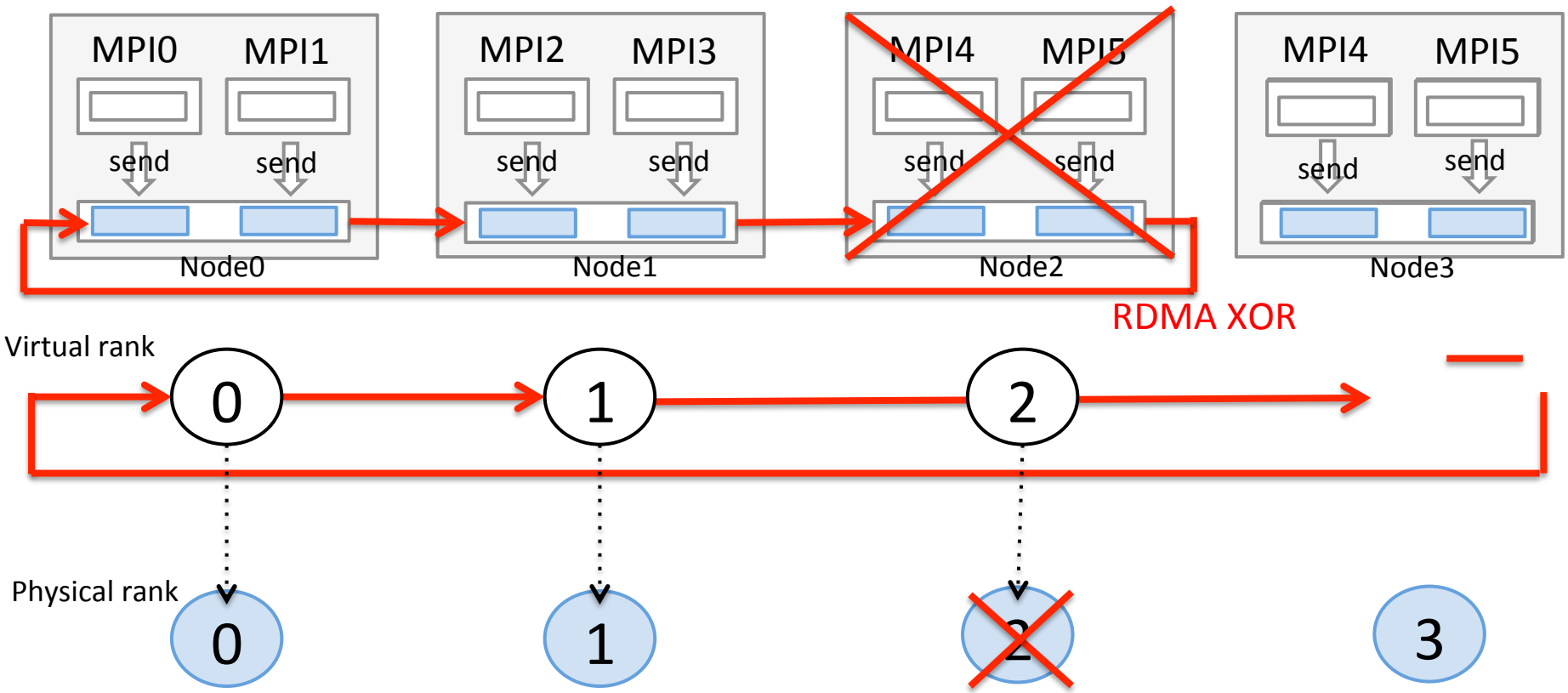
1. On a checkpoint, send to a staging client, then…
   - Staging nodes start XOR encoding in the background
   - Running processes can continue to run the computation
2. On a failure, choose an available node

System overview

1. On a checkpoint, send to a staging client, then…
   - Staging nodes start XOR encoding in the background
   - Running processes can continue to run the computation
2. On a failure, choose an available node
3. Restore lost checkpoints

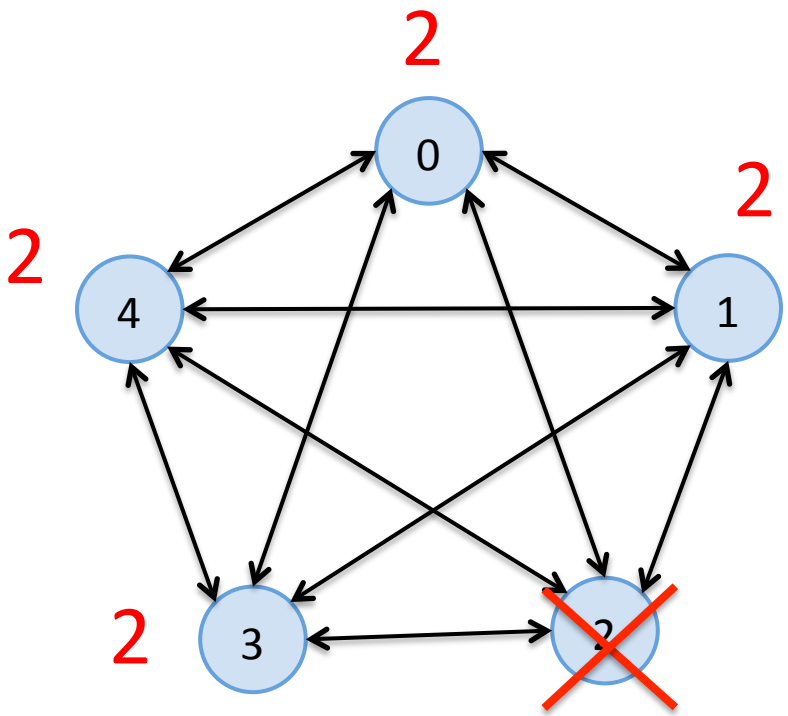
Rank re-mapping

- After restart, staging clients can not communicate with failed staging client
  - Staging clients communicate using virtual rank
  - On a failure, staging nodes update rank mapping between virtual and physical rank
  - By using virtual ranks, staging clients can continue to communicate each other with same ranks



Consistent process mapping

- Use of All-to-all connection
  - Current implementation makes All-to-all connection to be able to detect process failure, and update a consistent process mapping across staging processes

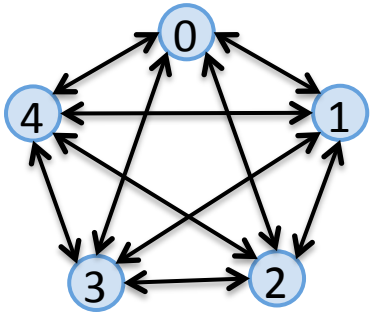


FMI: Fault tolerant Messaging Interface

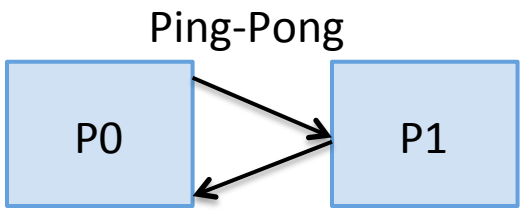
- ibverbs: Infiniband user level communication library
  - On a failure, the rest of processes can continue to run and communicate
- Primitive communication function using RDMA
  - FMI_Init(…)
  - FMI_{Isend|Irecv|Send|Recv}(…)
  - FMI_Wait(…)
  - FMI_Finalize (…)
- Some common optimizations
  - Use of Eager buffer
  - Use mutex to wait receiving message to minimize CPU usage (not use of busy wait

Preliminary experiment

- FMI Initialization
  - Scalability investigation



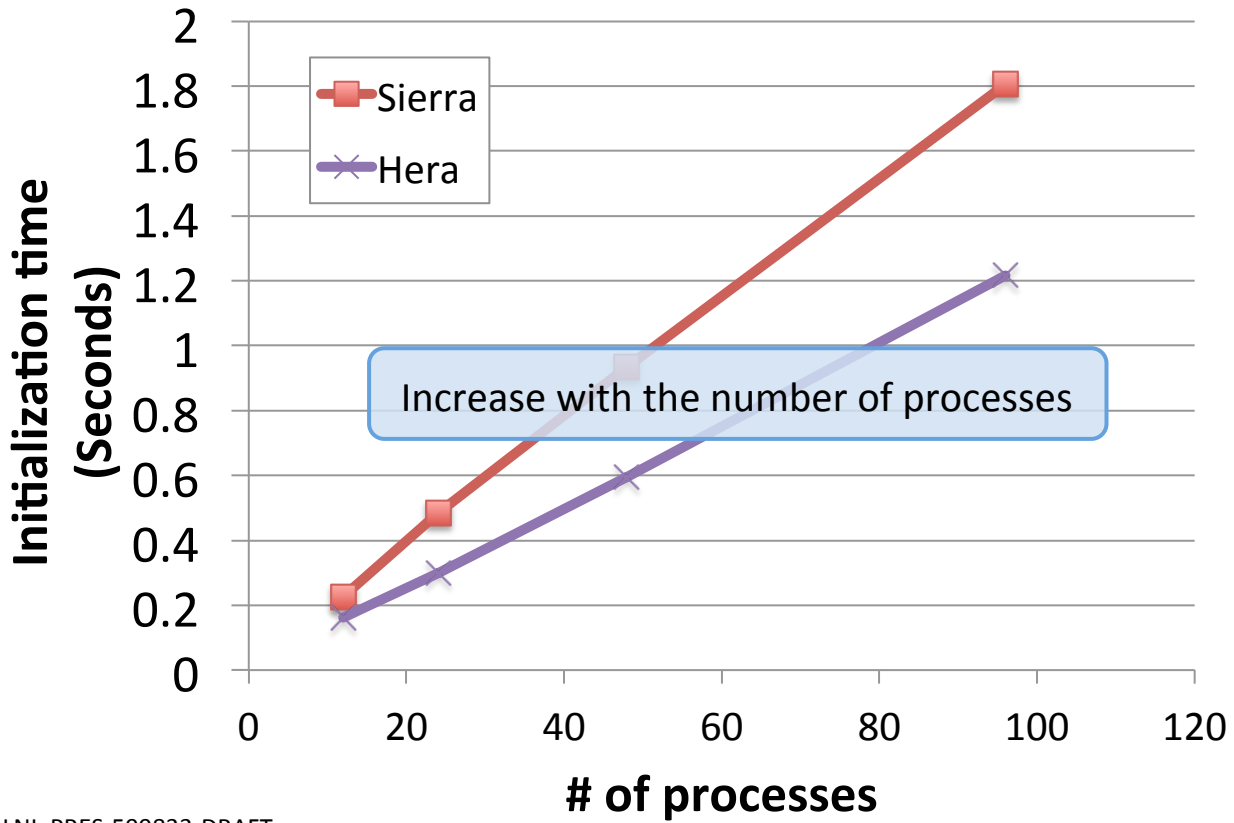
- FMI Performance
  - Ping-Pong: FMI vs. MPI

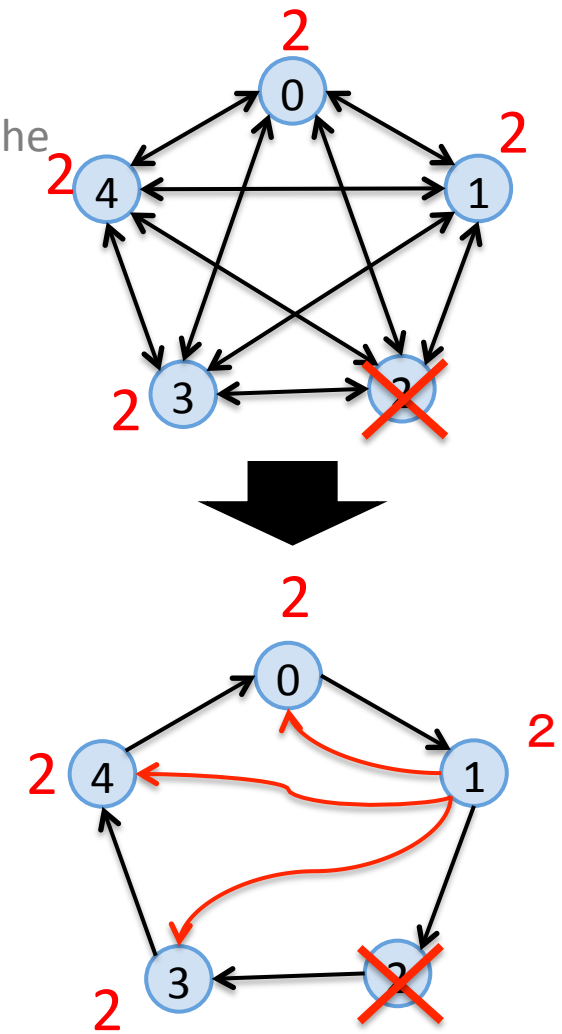


- XOR checkpointing performance using FMI
  - Scalability investigation

FMI Initialization

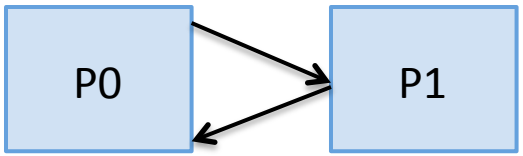
- Expected to increase to 28.8 with 1536 processes
  - Change from All-to-all connection to Ring one
  - On a failure, rank which detected failure broad cast the failed rank

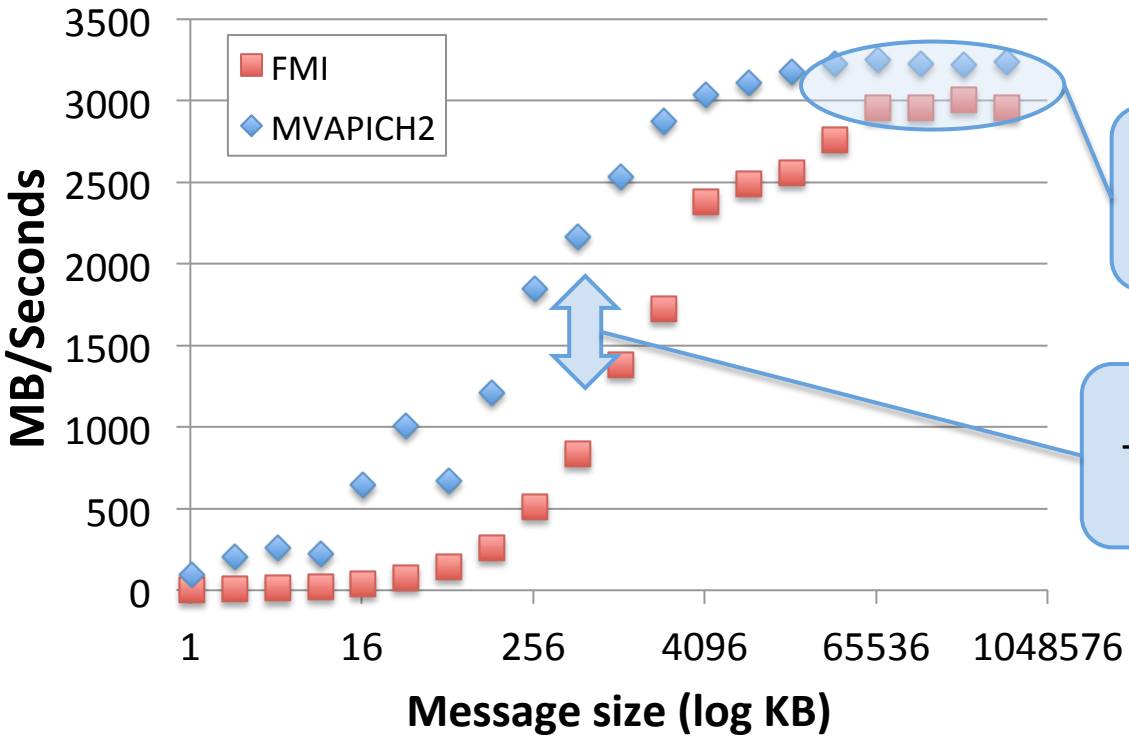




FMI Throughput on Sierra

- Pingpong between 2 processes
  - Sierra => QLogic HCA



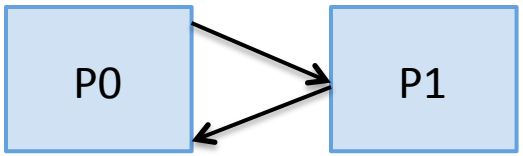


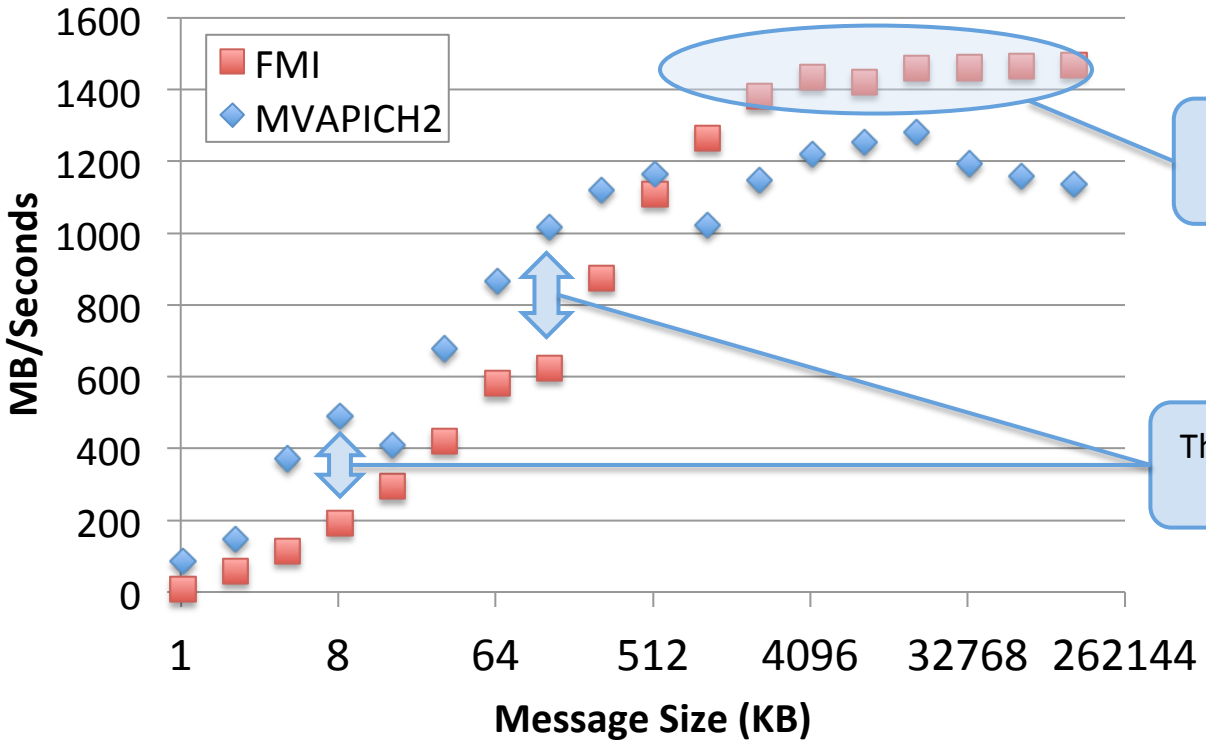
With large message size, FMI can achieve the almost same performance

There is performance big gap.
This is because ibverbs operations are emulated, which causes big latency

FMI Throughput on Hera

- Pingpong between 2 processes
  - Hera => Mellanox HCA



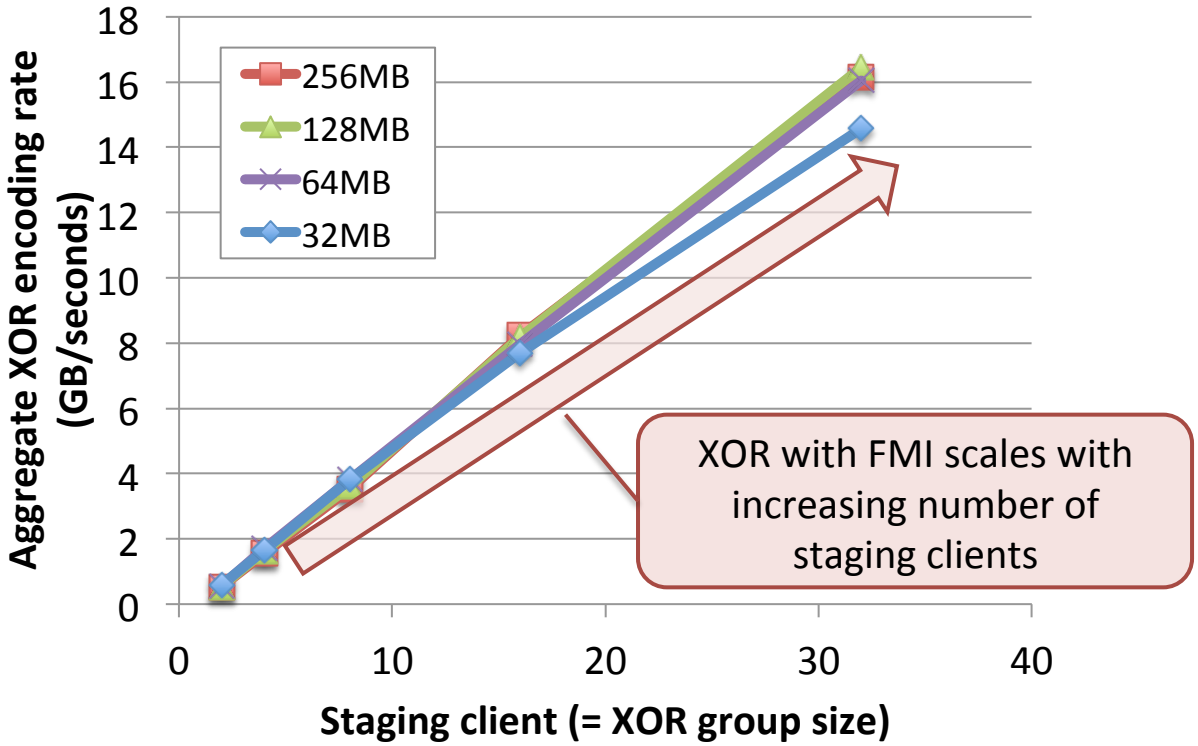


With large message size, FMI achieve higher performance that MPI

The performance gap decrease, because Mellanox HCA support ibverbs,

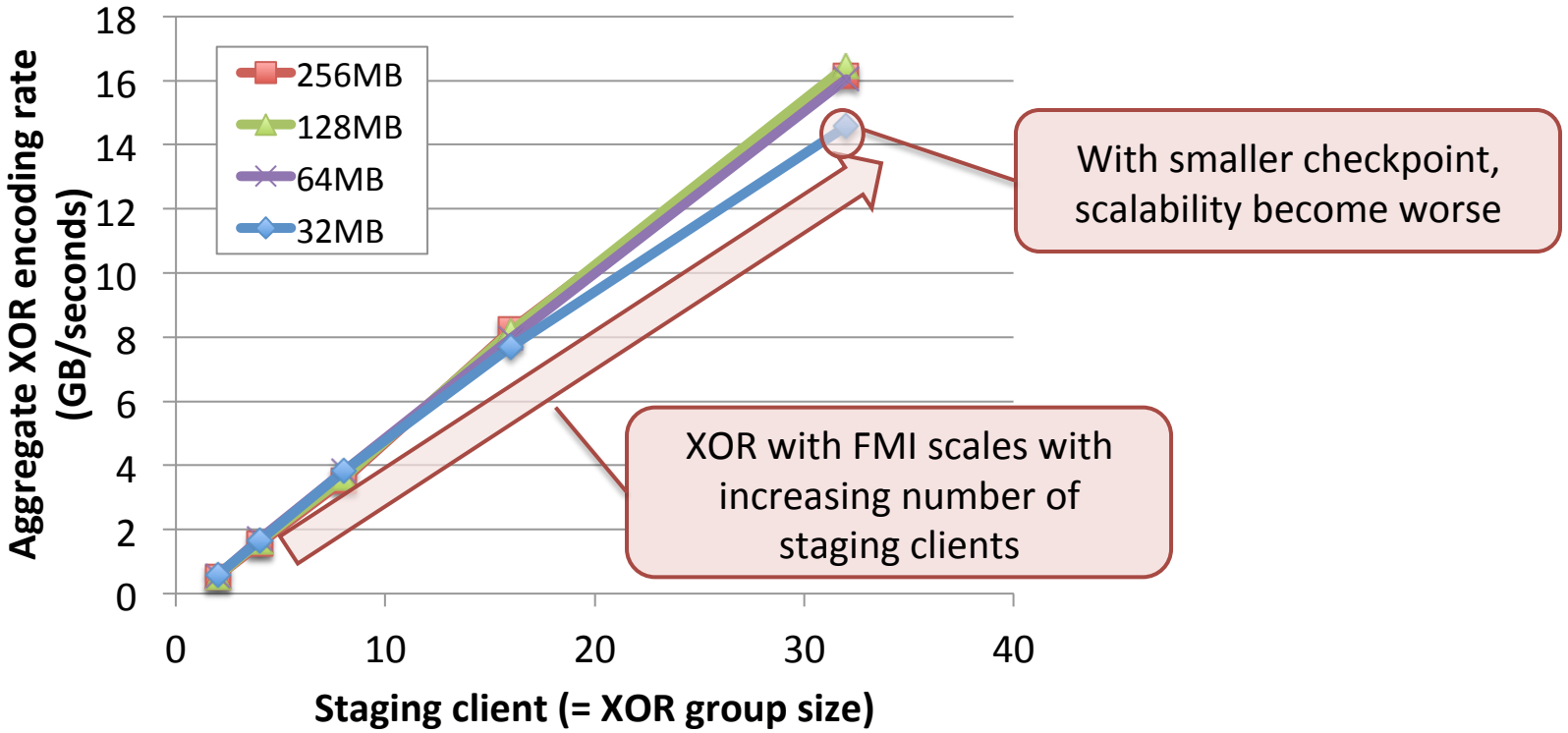
Aggregate XOR performance on Sierra

- XOR encoding time in different checkpoint size with increasing staging clients



Aggregate XOR performance on Sierra

- XOR encoding time in different checkpoint size with increasing staging clients

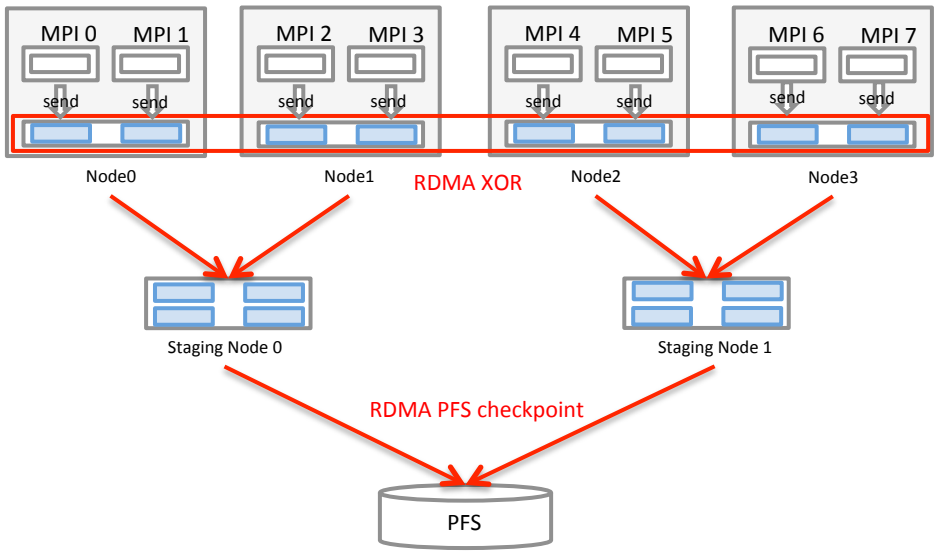


Conclusions

- Developed fault tolerant massaging library (FMI)
  - Achieve high throughput as MPI with large message size
- Developed XOR checkpointing using RDMA
  - Confirmed the XOR checkpointing is scalable

Future work

- Improvement of FMI
  - Change All-to-all connection to Ring or else
- Adaptive interval on a restart
- Integration with RDMA PFS checkpoint

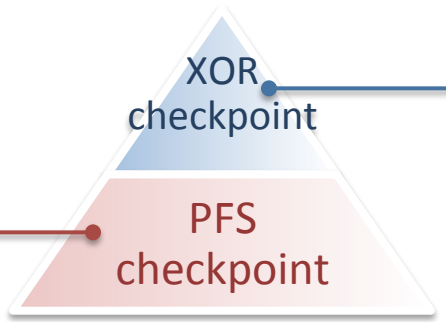


Summary: Efficient non-blocking MLC

- I -

Design and Modeling of a Non-blocking Checkpointing System

High efficiency on current and future systems with low PFS throughput requirement



Scalable RDMA XOR checkpoint

- II -

Towards Asynchronous and Adaptive Multi-level Checkpoint/Restart

Q & A

Speaker:

Kento Sato (佐藤 賢斗)
Tokyo Institute of Technology (Tokyo Tech)
kent@matsulab.is.titech.ac.jp
http://matsu-www.is.titech.ac.jp/~kent/index_en.html