RUTGERS
THE STATE UNIVERSITY
OF NEW JERSEY

RDI²

SAND2016-1864C

Fenix:
An Online Failure Recovery Library for MPI applications on top of ULFM

Marc Gamell[1], Keita Teranishi[2],
Rob Van Der Wijngaart[3], Manish Parashar[1]

[1] Rutgers Discovery Informatics Institute (RDI²), Rutgers University
[2] Sandia National Laboratories
[3] Intel Corporation

Based on:
"Exploring Automatic, Online Failure Recovery for Scientific Applications at Extreme Scales", SC14
Marc Gamell, Daniel S. Katz, Hemanth Kolla, Jacqueline Chen, Scott Klasky, Manish Parashar

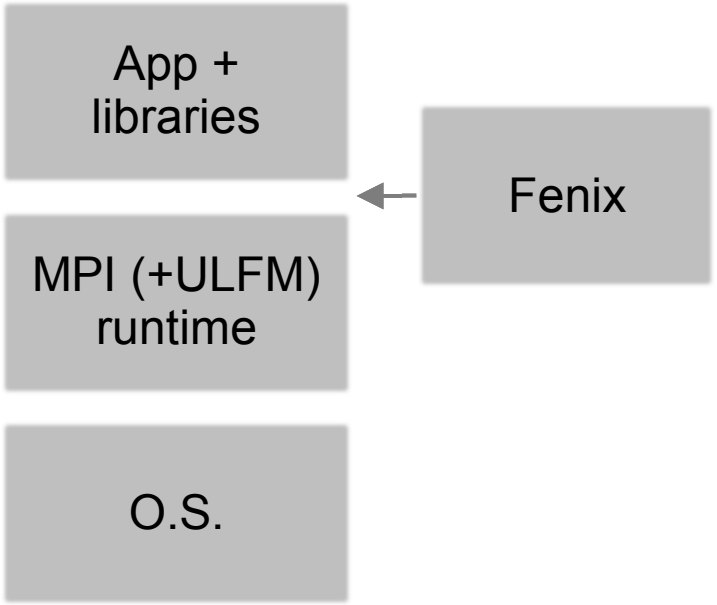
Introduction / Goal

- Fenix can be used effectively, efficiently, and productively to provide online fault tolerance to a major production code (S3D)

- How can an application use Fenix?
- How does Fenix use ULFM and PMPI?

Table of Contents

Key Contributions



Implementation details

- Built on top of **ULFM**
- Tested up to
  - 16384 cores w/ failures
  - 250k cores w/o failures
- Available for C and Fortran applications

Approach

- Targets **MPI-based** parallel applications
- Offers two disjoint interfaces

Fenix Interfaces:

1. Process/rank recovery

- **Online, semi-transparent recovery from process**, node, blade and cabinet **failures**
- **Tolerates a variety of MTBFs**
  - **even extreme MTBFs of <1 minute**

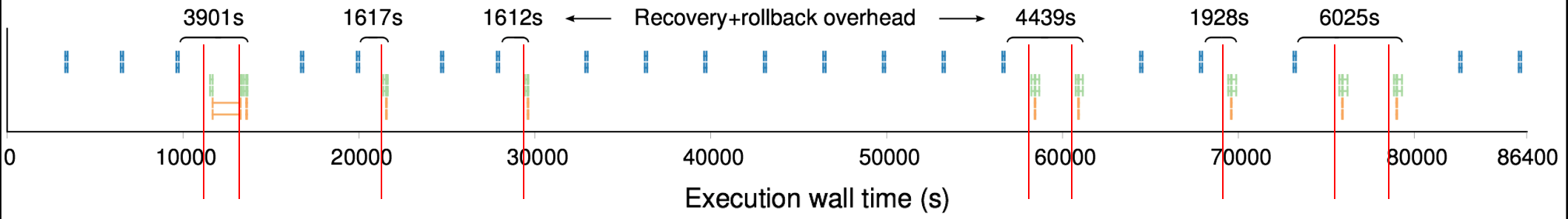
2. Data recovery

- Uses **application-specific, double in-memory, implicitly coordinated checkpoints**

Experimental Evaluation

- Deployed Fenix on **Titan** Cray XK7 at ORNL
- **S3D** combustion numerical simulation
- Sustained performance with **MTBF = 47 seconds**
- Experiments inject real process failures (SIGKILL)

Motivating Use Case – S3D production runs



- 24-hour tests using Titan (125k cores)
- 9 process/node failures over 24 hours
- Failures are promoted to **job failures**, causing all 125k processes to exit
- Checkpoint (5.2 MB/core) has to be done to the PFS

		Total cost
Checkpoint (per timestep)	**55 s**	1.72 %
Restarting processes	**470 s**	5.67 %
Loading checkpoint	**44 s**	1.38 %
Rollback overhead	**1654 s**	22.63 %
Total overhead		**31.40 %**

Motivating Use Case – Possible solution

- Process failures cannot be promoted to job failures, to:
  - **Reduce recovery** cost
  - Keep process memory (contains checkpoints)

Online recovery

- **Checkpoint frequency** has to be dramatically **increased**, e.g.

Checkpoint application-specific data in process memory w/o coordination

ULFM – User Level Failure Mitigation

User Level Failure Mitigation is a **set of MPI extensions to report errors, provide interfaces to stabilize the distributed state, and restore the communication capabilities in applications affected by process failures.** Relevant communicators, RMA windows and I/O files can be reconstructed online, without restarting the application, as required by the user recovery strategy.

FLEXIBILITY

- No particular recovery model is imposed or favored. Instead, a set of versatile APIs is included that provides support for different recovery styles (checkpoint, ABFT, iterative, Master-Worker, etc.).
- Application directs the recovery, it pays only for the level of protection it needs.
- Recovery can be restricted to a subgroup, preserving scalability and easing the composition of libraries.

PERFORMANCE

- Protective actions are outside of critical MPI routines.
- MPI implementors can uphold communication, collective, one-sided and I/O management algorithms unmodified.
- Encourages programs to be reactive to failures, cost manifests only at recovery.

PRODUCTIVITY

- Backward compatible with legacy, fragile applications.
- Simple and familiar concepts to repair MPI.
- Portability guaranteed by standardization.
- Provides key MPI concepts to enable FT support from library, runtime and language extensions.

Excerpt of ULFM flyer at SC15 by ICL@UTK

Two Philosophy Changes for the Application

Fenix:

- Provides semi-transparent online recovery to particular application types
- Implements a particular recovery mode thanks to generic ideas behind ULFM

Application view:

- Offer a **single point** in which all ranks will **return** upon failure has been recovered
  - longjump
- Application should use **Fenix's resilient communicator** instead of MPI_COMM_WORLD
  - Resilient communicator is created when initializing Fenix
  - All communicators derived from it can also be used

What about other libraries?

- If they do not use Fenix (and keep state): teardown and re-initialize
- If they use Fenix themselves:
  - Not supported for the moment
  - Maybe libraries can register callbacks to be called upon recovery

Fenix – Failure Recovery Interface

```
void Fenix_Init ( int *status,
                  MPI_Comm comm,
                  MPI_Comm *newcomm,
                  int *argc, int ***argv,
                  int num_spare_ranks,
                  Fenix_Comm_repair_policy repair_policy,
                  int *error);
```

INITIAL_RANK
RECOVERED_RANK
SURVIVOR_RANK

MPI_COMM_WORLD

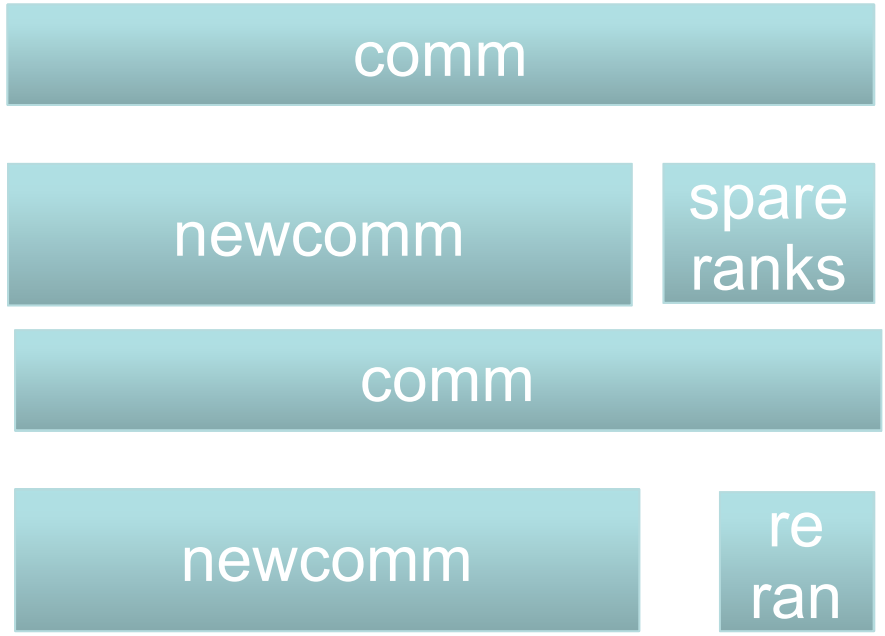
App should use newcomm instead of MPI_COMM_WORLD

NO_SPAWN
SPAWN

```
void Fenix_Finalize ( );
```

newcomm: resilient communicator

- Non-shrinking: SPAWN policy
- Shrinking: NO_SPAWN + num_spare_ranks = 0
- Mixed (only spares): NO_SPAWN policy + num_spare_ranks > 0



Fenix + ULFM – Recovery Stages

1. Failure detection

- With ULFM, MPI communicating calls may return failure codes
- Fenix captures them using MPI profiling interface
  - ☹ For now, this implies **no tools can be used in conjunction with Fenix**
  - ☺ In the future, maybe **MPI will replace PMPI with a method that allows attaching multiple tools (QMPI/MPI Extension Interface)?**
- Uses MPI_Comm_revoke to spread notification

2. Environment recovery

- Repair only main communicator
  - Non-shrinking model:
    - Use spare process pool and
    - Re-spawn processes
  - **Non-shrinking up to a certain number of failures**
    - **Use spare process pool**
  - Shrinking model
- Re-creation of user communicators: done by the user

→ This is shown in the following slides

Fenix + ULFM – Internal Initialization Pseudocode

Assume we have two communicators:

- `MPI_Comm comm` (originally, a dup from `Fenix_Init`'s input `comm`)
- `MPI_Comm newcomm` (repaired communicator to be used, returned as `newcomm` from `Fenix_Init`)

```
fenix_internal_init() {
   MPI_Comm_set_errhandler(comm, MPI_ERRORS_RETURN);
   MPI_Comm_rank(comm, &rank);
   MPI_Comm_rank(comm, &size);

   if(rank >= size – num_of_spare_ranks) {  // SPARE RANK
      PMPI_Comm_split(comm, MPI_UNDEFINED, rank, &newcomm);

      for(;;) {  // wait for failure to occur
         ret = MPI_Recv(..., MPI_ANY_SOURCE, MPI_ANY_TAG, comm);
         if(ret == MPI_SUCCESS)
            exit();
      }

   } else {  // NOT A SPARE RANK
      PMPI_Comm_split(comm, 0, rank, &newcomm);
   }
}
```

Fenix + ULFM – Failure Detection Pseudocode

- Override MPI functions and use PMPI interface, e.g.

```
int MPI_Send(void *buf, int count, MPI_Datatype datatype,
             int dest, int tag, MPI_Comm comm)
{
   int ret;
   ret = PMPI_Send(buf,count,datatype,dest,tag,comm);
   fenix_internal_test_return_code(ret);
   return ret;
}
```

Fenix + ULFM – Testing for Failures Pseudocode

```
void fenix_internal_test_return_code(int ret) {
   switch(ret) {
      case MPI_SUCCESS:
         return;
      case MPIX_ERR_PROC_FAILED:
         MPIX_Comm_revoke(comm);
         MPIX_Comm_revoke(newcomm);
         fenix_internal_revoke_user_communicators();
      case MPIX_ERR_REVOKED:
         fenix_internal_communicator_repair();
         break;
   }
}
```

Fenix + ULFM – Recovery Pseudocode (1/2)

```
fenix_internal_communicator_repair() {
   MPI_Comm comm_shrink;
   MPIX_Comm_shrink(comm, &comm_shrink);

   // Do we have enough spare ranks?
   MPI_Comm_size(comm, &old_comm_size);
   MPI_Comm_size(comm_shrink, &new_comm_size);
   ranks_needed = old_comm_size - new_comm_size;
   if(num_of_spare_ranks < ranks_needed)
        // for today's discussion, consider an error

   // Which ranks failed?
   MPI_Comm_rank(comm, &old_rank);
   PMPI_Allgather(&old_rank, 1, MPI_INT,
                  survivor_ranks, 1, MPI_INT, comm_shrink);
   // locally determine the failed ranks by using the survivor_ranks array

   ...
```

Fenix + ULFM – Recovery Pseudocode (2/2)

```
fenix_internal_communicator_repair() {
    // Shrink the communicator
    // Do we have enough spare ranks?
    // Which ranks failed?
    ...

    // Assign spare ranks to failed ranks
    if(old_rank >= size_newcomm)  // WAS A SPARE RANK
       if((old_comm_size-1-old_rank) < procs_needed) // RECOVERED RANK
          old_rank = // pick one of the failed ranks
    else  // WAS NOT A SPARE RANK
       PMPI_Comm_free(&newcomm);
    num_of_spare_ranks -= num_failed_ranks;

    // Re-name the spare ranks in the new communicator
    PMPI_Comm_free(&comm);
    PMPI_Comm_split(comm_shrink, 0, old_rank, &comm);
    PMPI_Comm_free(&comm_shrink);

    fenix_internal_init();

    // no matter who called fenix_internal_communicator_repair, return to Fenix_Init()
}
```

Which communicators does Fenix revoke?

- When failure is detected by a rank, ALL communicators derived from `newcomm` are revoked
  - The only communicators that the application should use with Fenix philosophy
- Derived communicators are detected by the profiling interface
  - MPI_Comm_split, MPI_Comm_create, MPI_Comm_dup…
  - Captured by profiling interface when they are first derived
  - Upon failure, revoked

Fenix – Recovery Stages

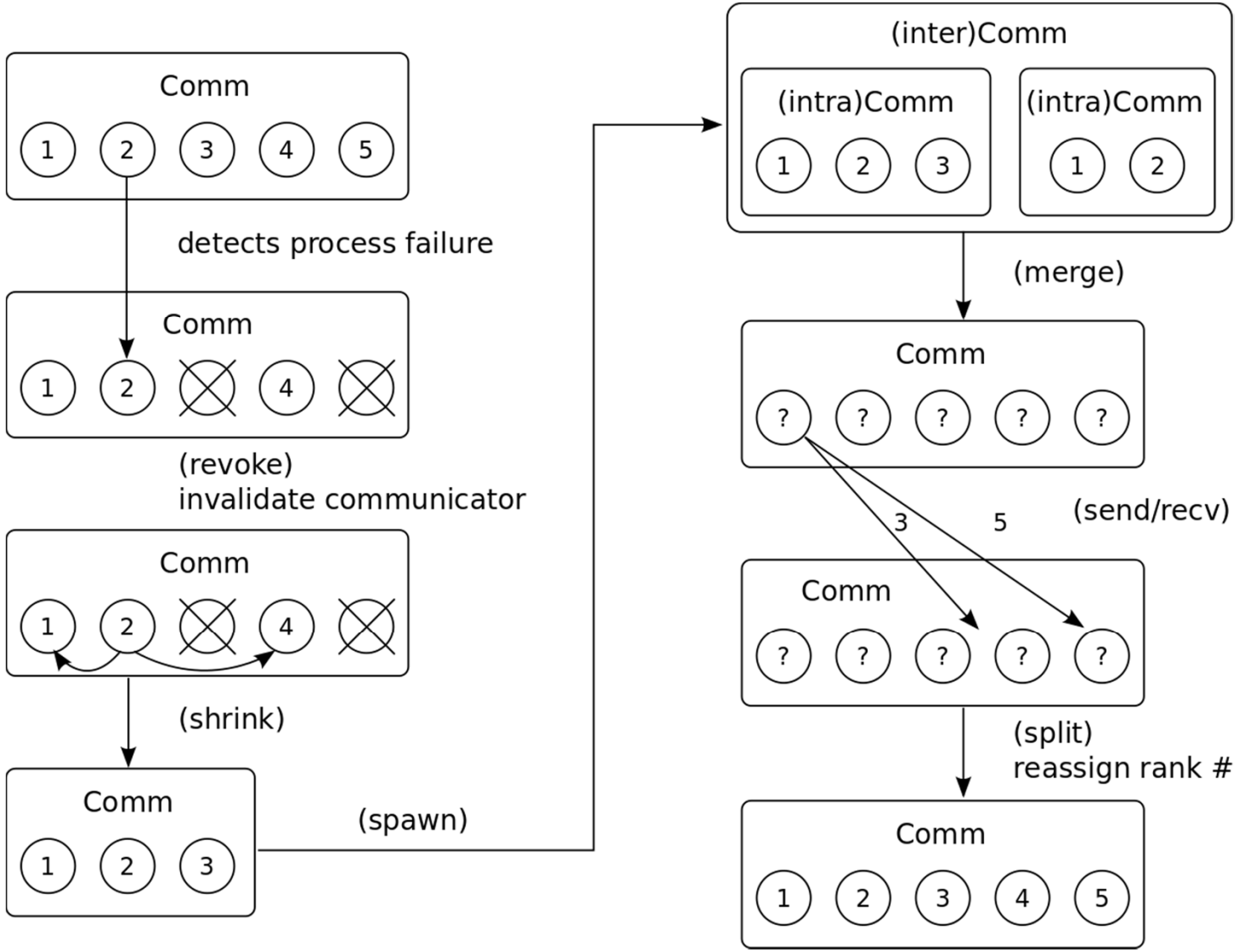
1. Failure detection

Based on

- ULFM return codes
- MPI profiling interface
- Comm revoke

2. Environment recovery

- Re-spawn / Spare process pool
- Repair “newcomm” communicator
- Delay re-creation of user comms



“Respawn” recovery mode depicted (similar for spare process pool)

Table of Contents

Core of Fenix Data Recovery Interface

```
Fenix_Checkpoint ( int member_id,
                   void *buffer,
                   int count,
                   MPI_Datatype datatype,
                   Fenix_Checkpoint_group group,
                   Fenix_Checkpoint_subset subset);

Fenix_Checkpoint_commit (
                   int *index,
                   Fenix_Checkpoint_group group);
```

Table of Contents

Fenix – Simple Usage Example

Assume work1() and work2() use an MPI communicator to communicate with other ranks

```c
/* Non-fault-tolerant version */
int main()
{
  int it;
  int A[100], B[50];

  initialize(A, B);

  for(it=0 ; it<1000 ; it++) {
    work1(A, MPI_COMM_WORLD);
    if(A[0] > 200) {
      work2(A, B, MPI_COMM_WORLD);
    }
  }
}
```

```c
/* Fault tolerant version with Fenix */
int main()
{
  int it;
  int A[100], B[50];
  int status;
  MPI_Comm new_comm_world;

  Fenix_Init(&status, MPI_COMM_WORLD, &new_comm_world,
             ...,
             10, // num_spare_ranks
             FENIX_COMM_REPAIR_POLICY_NO_SPAWN, // repair_policy
             ...);
  if( status == FENIX_STATUS_INITIAL_RANK ) {
    /* no failure occurred */
    it = 0;
    initialize(A, B);
    Fenix_Checkpoint(990, &it, 1, MPI_INT);
    Fenix_Checkpoint(991, A, 100, MPI_INT);
    Fenix_Checkpoint(992, B, 50, MPI_INT);
    Fenix_Checkpoint_commit();
  } else {
    /* ranks recovered from a failure, now restore data */
    Fenix_Checkpoint_restore(990, &it, 1, MPI_INT,
        FENIX_LATEST_COMMIT);
    Fenix_Checkpoint_restore(991, A, 100, MPI_INT,
        FENIX_LATEST_COMMIT);
    Fenix_Checkpoint_restore(992, B, 50, MPI_INT,
        FENIX_LATEST_COMMIT);
  }

  for( ; it<1000 ; it++) {
    Fenix_Checkpoint(990);
    work1(A, new_comm_world);
    if(A[0] > 200) {
      work2(A, B, new_comm_world);
      Fenix_Checkpoint(992);
    }
    Fenix_Checkpoint(991);
    Fenix_Checkpoint_commit();
  }
}
```

Rank failure line 34 (due to imbalance, some ranks are in it=102, the rest are in it=103 or it=104)

Fenix will resume the execution, returning from Fenix_Init()

This sets it=101, and A, B have the same values they had at iteration 101

S3D Modifications

Original vs Fenix-enabled

← Main function

Only 35 new, changed, or rearranged lines in S3D code

Topology module

↓

vs Fenix-enabled

Table of Contents

Goal and Methodology

- Experiment:

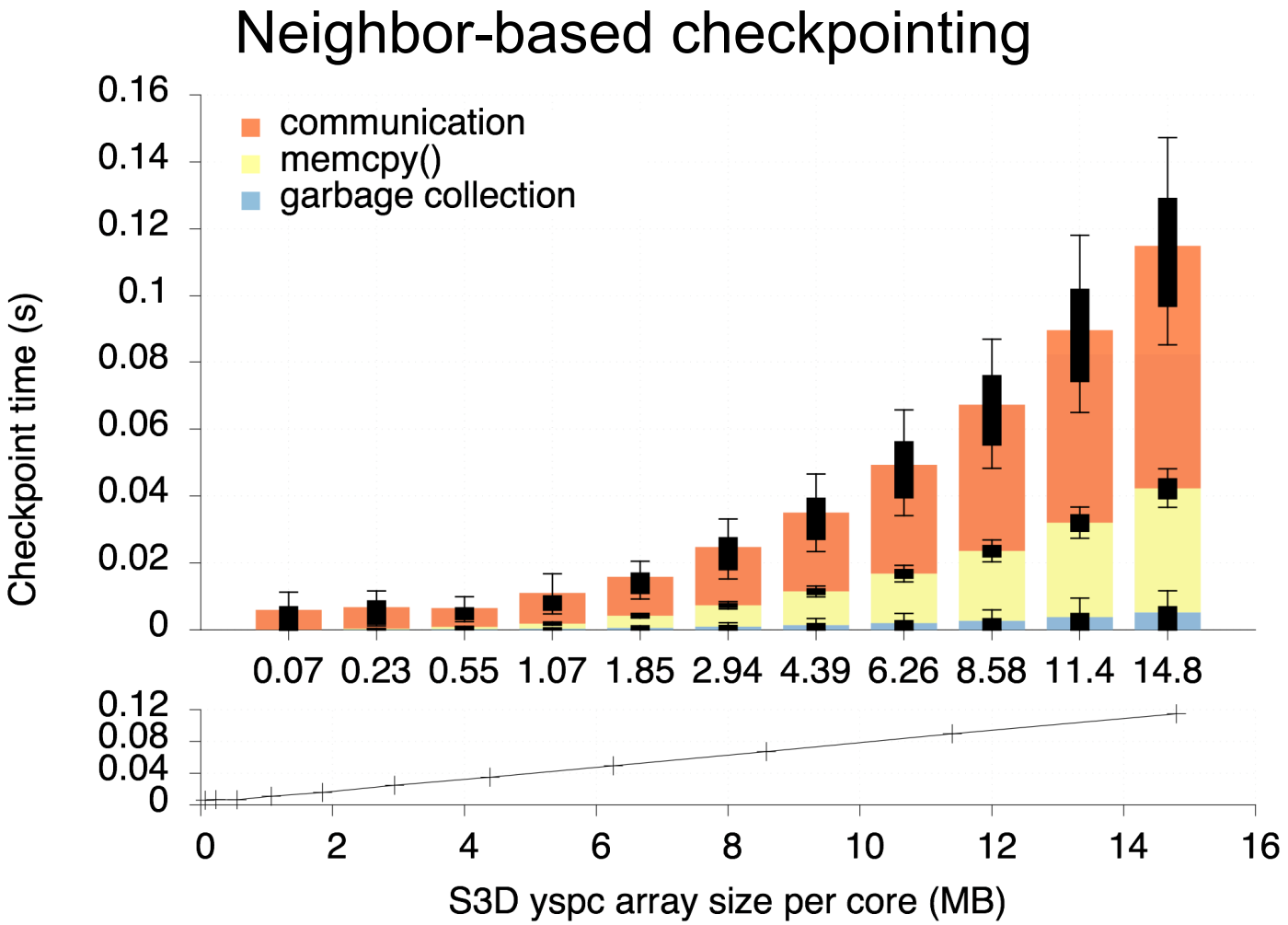
S3D execution mimicking a future extreme-scale scenario

- Injecting failures every (MTBF):
  - 47 seconds
  - 94 seconds
  - 189 seconds
- Checkpoint 8.58 MB/core

- Procedure:
  1. Evaluate checkpoint scalability
  2. Calculate and validate optimal checkpoint interval
  3. Evaluate recovery scalability
  4. Run experiment

- Evaluation on ORNL Titan - Cray XK7

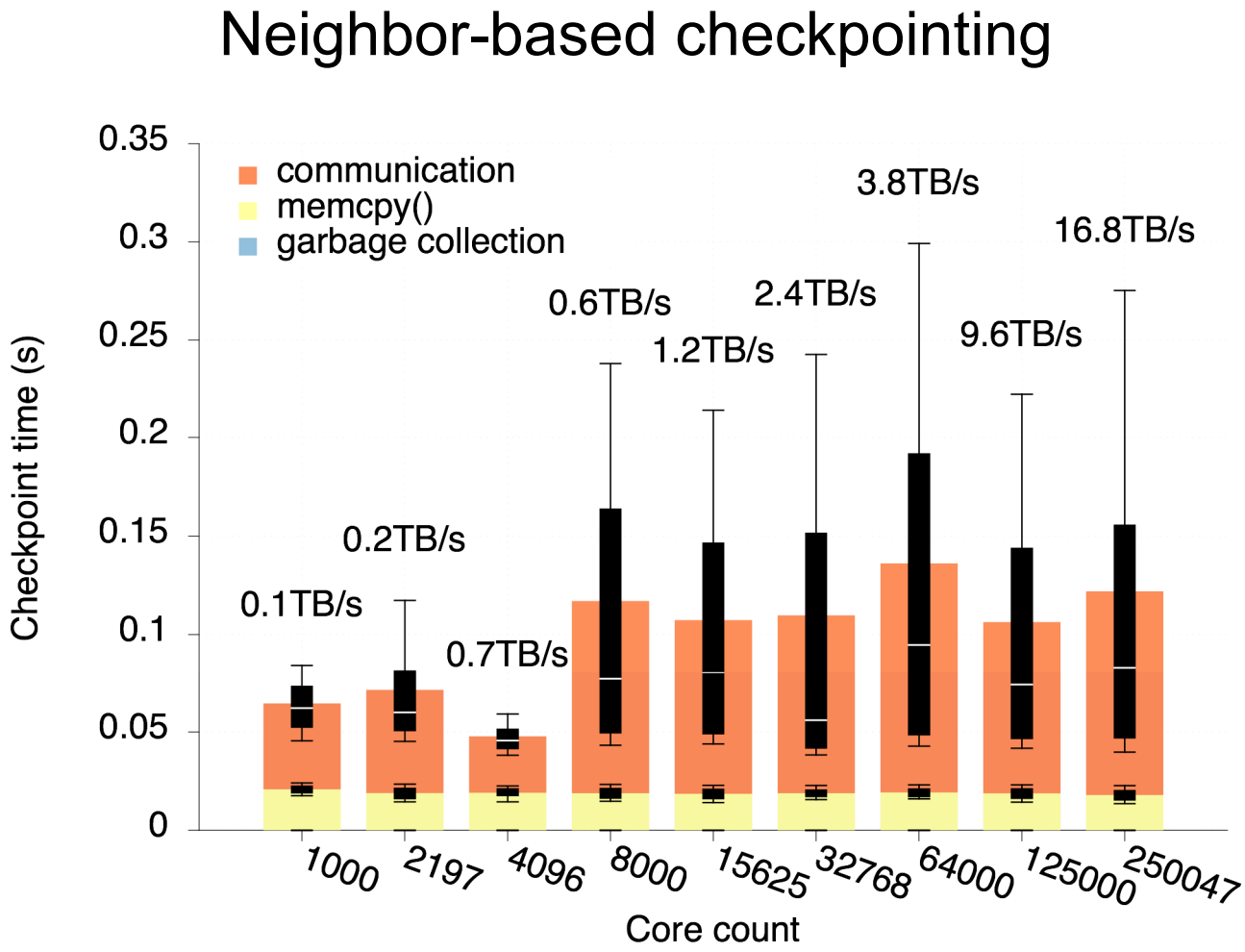
1. Failure-free **checkpoint cost (data size)**



Conclusions:

- Scales linearly with data size increase
- Huge communication cost

1. Failure-free **checkpoint cost (core count)**



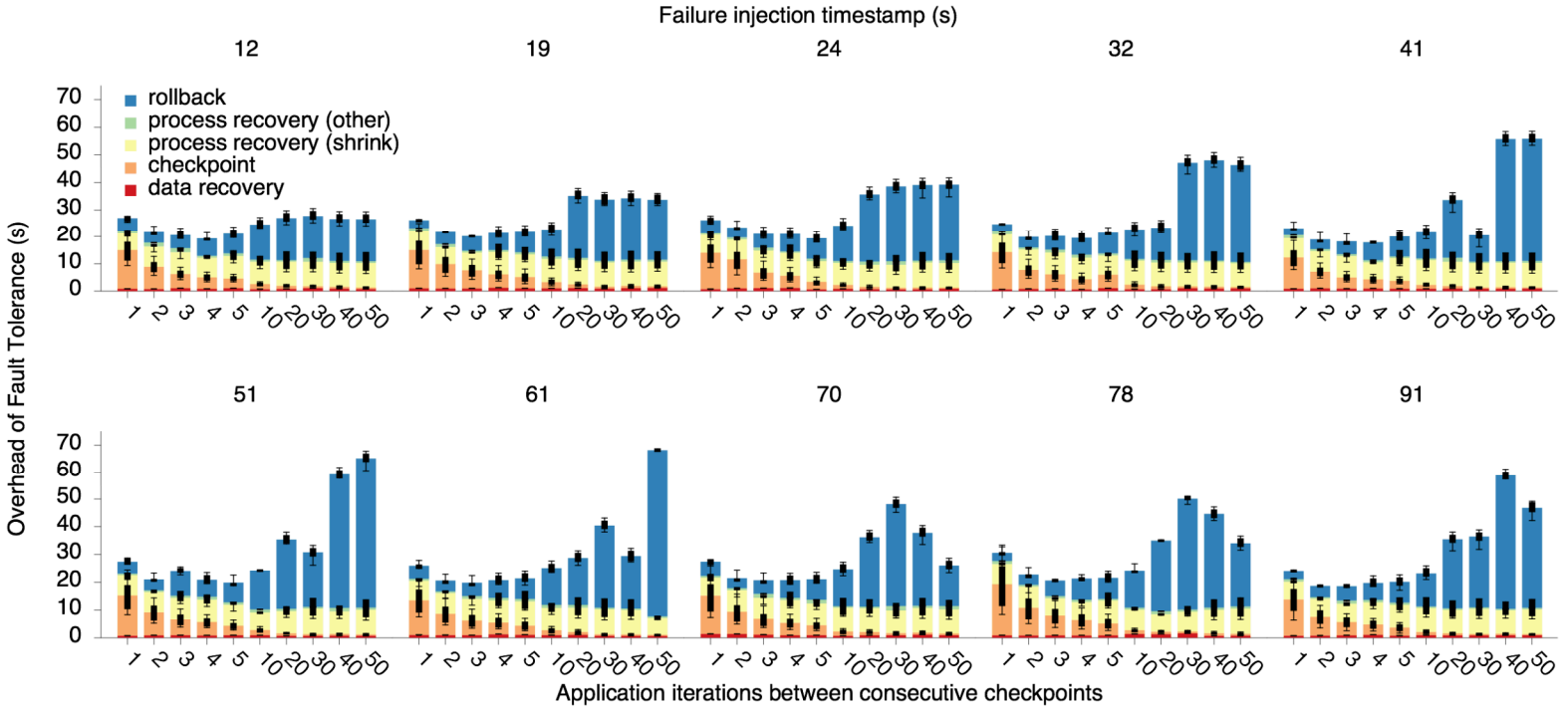
Conclusions:

- **Good scalability**
- **Really small footprint: O(0.1s)**

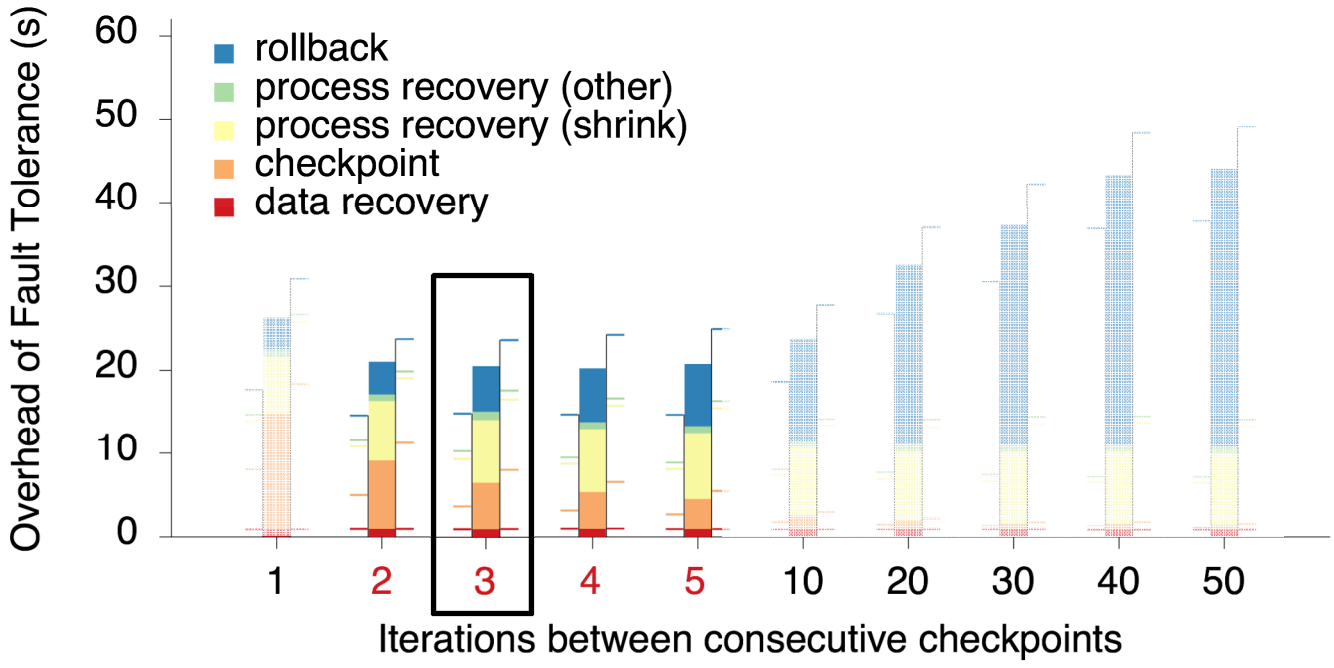
2. Optimal **checkpoint rate**

- Calculated by Young's formula:
  - T_S = checkpoint time = 0.0748 s (with 2197 cores)
  - T_F ≡ system's MTBF = {47, 94, 189} s (3 tests)
  - $T_C \equiv \sqrt{2 T_S T_F}$ = {2, 3, 4} S3D iterations

- Empirically validating the 94-second MTBF optimal rate:
  - Inject only one failure at a specific wall time within the first 94 seconds of a test
  - Repeat using:
    - Different checkpoint rates
    - Different failure injection timestamps
  - Optimal rate must offer smaller overall rollback cost!

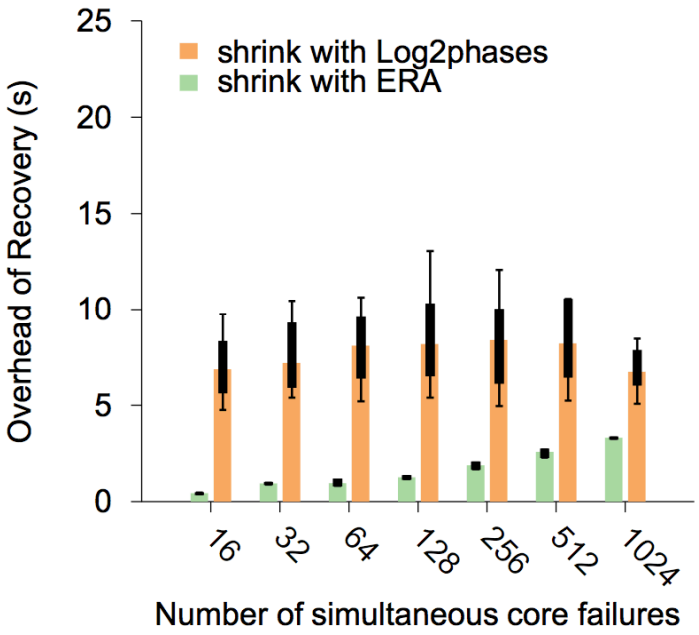


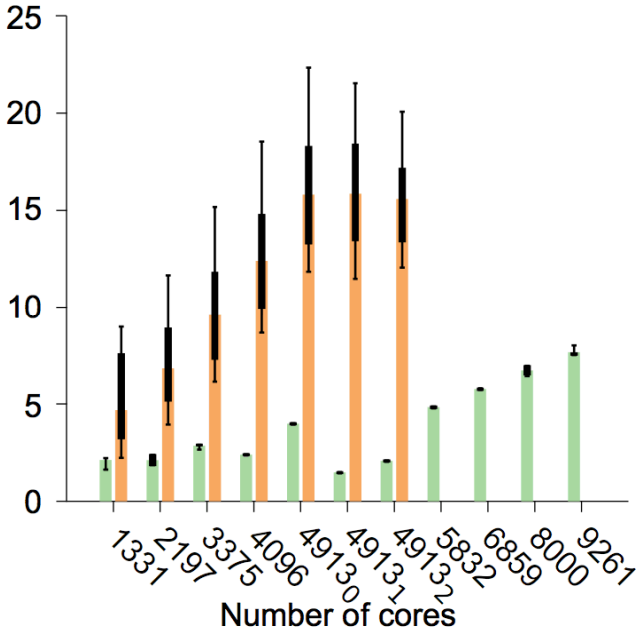
Average of all failure injection timestamps

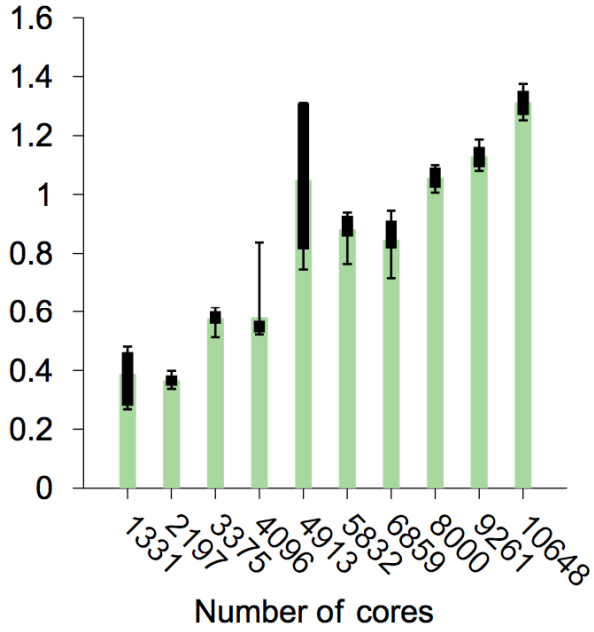


2 to 5 iterations is the optimal checkpoint period, validating Young's formula

3. Recovery overhead

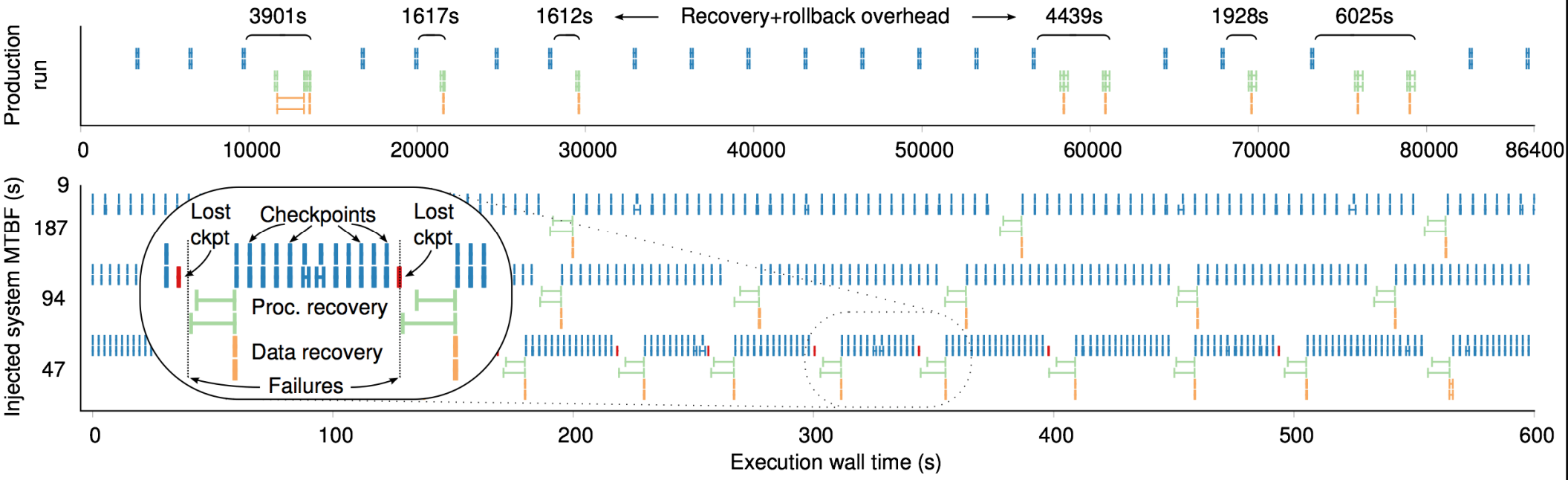






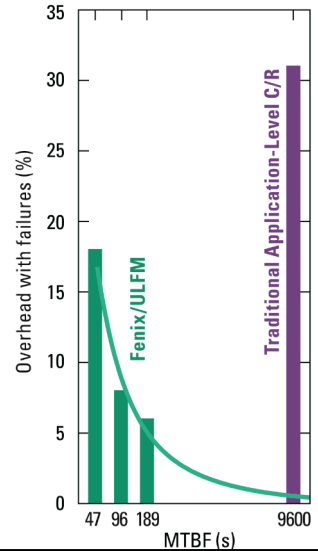
- 2197 cores
- 256-core failure (i.e. 16 nodes)
- 16-core failure (i.e. 1 nodes)

4. Recovering from **high-frequency failures**

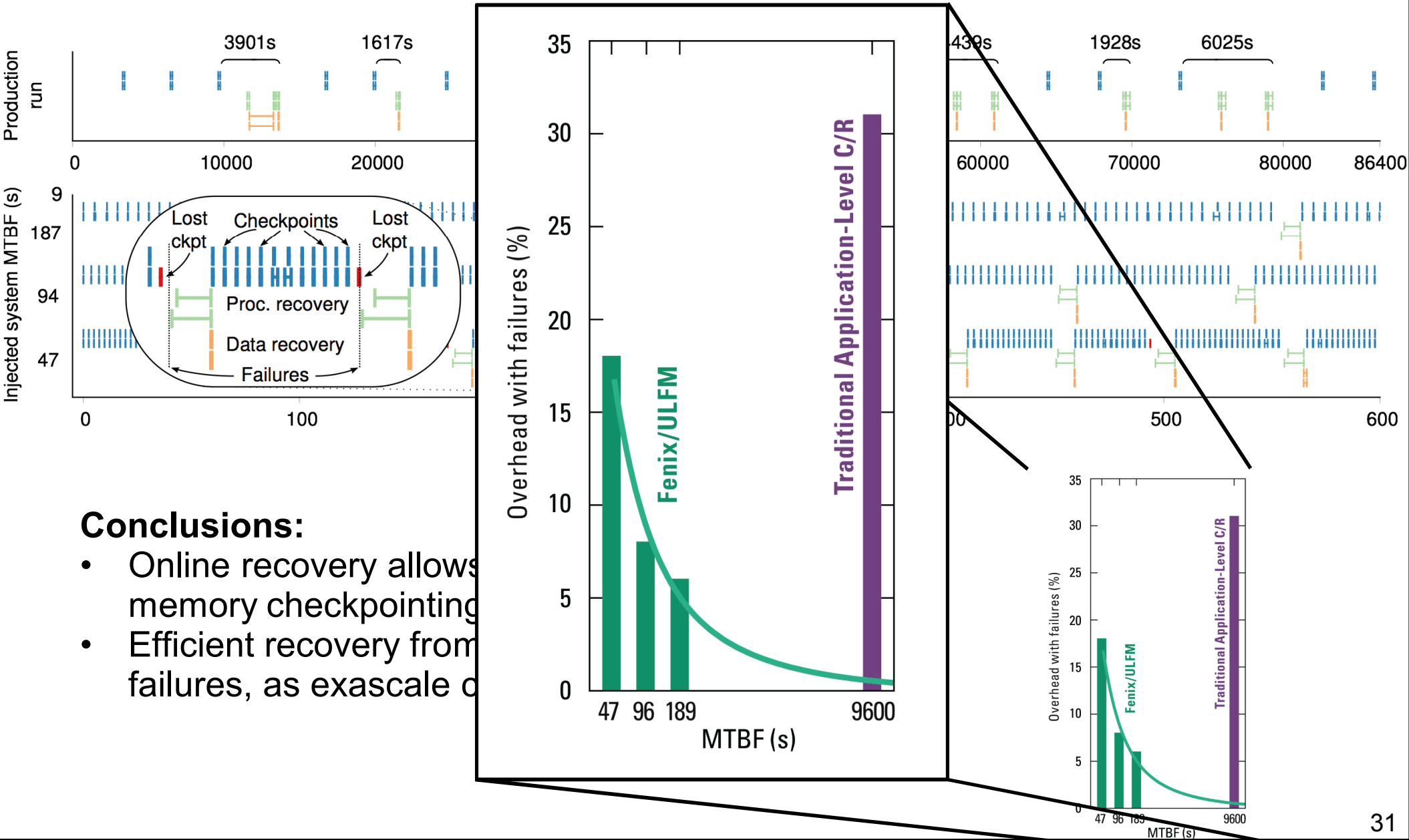


Conclusions:

- Online recovery allows the usage of in-memory checkpointing, O(0.1s)
- Efficient recovery from high-frequency node failures, as exascale compels



4. Recovering from **high-frequency failures**



Conclusions:

- Online recovery allows
  memory checkpointing
- Efficient recovery from
  failures, as exascale c

Table of Contents

Conclusion

- Application-awareness can help resilience at scale
- Fenix provides
  - **Online Failure Recovery (reducing failure overhead)**
  - (uses in-memory, application-specific**, high frequency checkpointing)**
  - **Simple API built on top of ULFM:** only 35 new, changed, or rearranged lines in S3D code
- Deployed and empirically tested on Titan Cray XK7
  - **S3D+Fenix tolerate failure rates <1 min** with lower overhead as coordinated CR with failure rates of ~2.5 hours

"Exploring Automatic, Online Failure Recovery for Scientific Applications at Extreme Scales", SC14
Marc Gamell, Daniel S. Katz, Hemanth Kolla, Jacqueline Chen, Scott Klasky, Manish Parashar

Thank you