Optimal Placement of Retry-Based Fault Recovery Annotations in HPC Applications

Exascale Co-Design Center for Materials in Extreme Environments

Ignacio Laguna, Martin Schulz, Jeff Keasler, David Richards, Jim Belak
Lawrence Livermore National Laboratory
Main contact: ilaguna@llnl.gov

Motivation

- Mean-time-between-failures may increase in future HPC systems
- We may not rely only on the checkpoint/restart approach
- Retry-based methods allow recovery by re-trying a code region
- Two retry-based approaches:

(1) Identify or create idempotent code (i.e., code where re-execution is free of side effects)
(2) Place retry annotations (involves micro-checkpointing data)
However, how do we place optimally these annotations?

Overview of Existing Fault Recovery Methods

Replication in *Space*

- Redundant multi-threading
- Redundant VMs
- Lockstepping

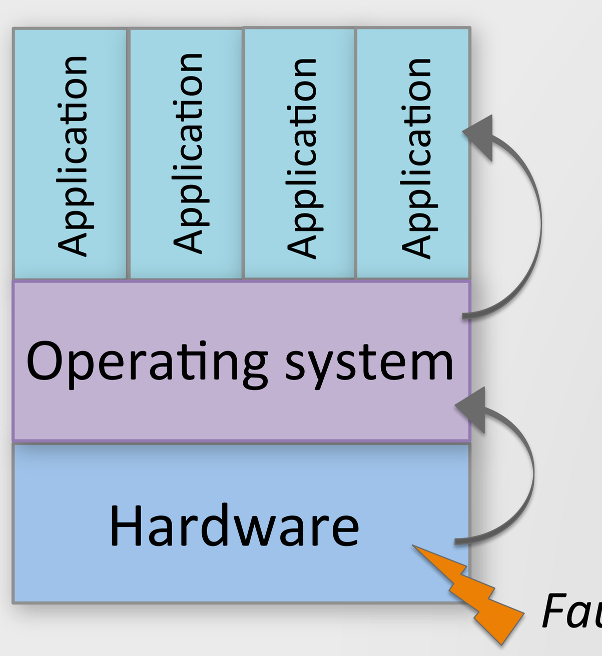
Incur hardware overheads
Inefficient for parallel codes

Replication in *Time*

- Checkpoint / Restart
- Retry idempotent code
- Micro-checkpoint / Retry

Assumptions for the Retry-Based Model

Replication in Time



- Fault detection in hardware
- Notifications of recoverable faults (e.g., via synchronous traps)
- Recovery at application level

Annotations are Required to Express Recovery Scope

- Programmer annotates (or protect) code block
- If a fault occurs, a code block is re-executed
- The decision on where to place annotations is critical or large overheads can occur
- No research work has evaluated how to optimally place these annotations

Original code

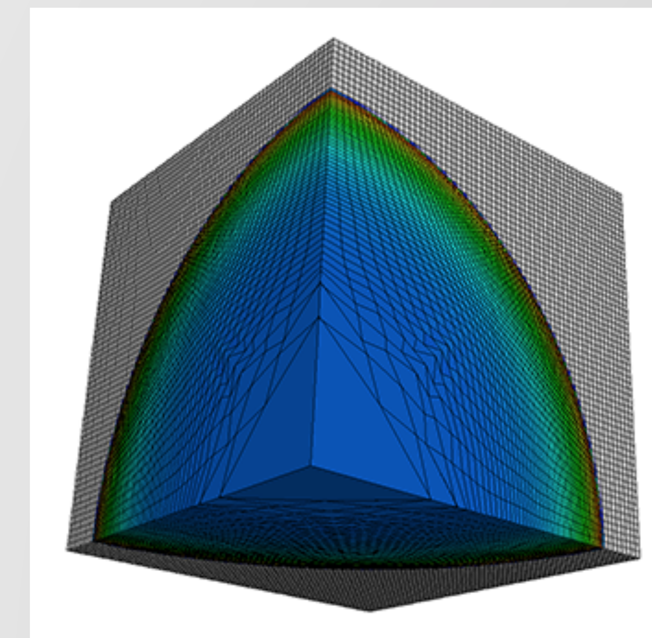
```
void function(double *array)
{
  for (...)
    array[i] = array[i-1] + ...
}
```

Annotated code

```
void function(double *array)
{
  RETRY{
    for (...)
      array[i] = array[i-1] + ...
  }
}
```

Sample Application: LULESH

Shock hydrodynamics problem

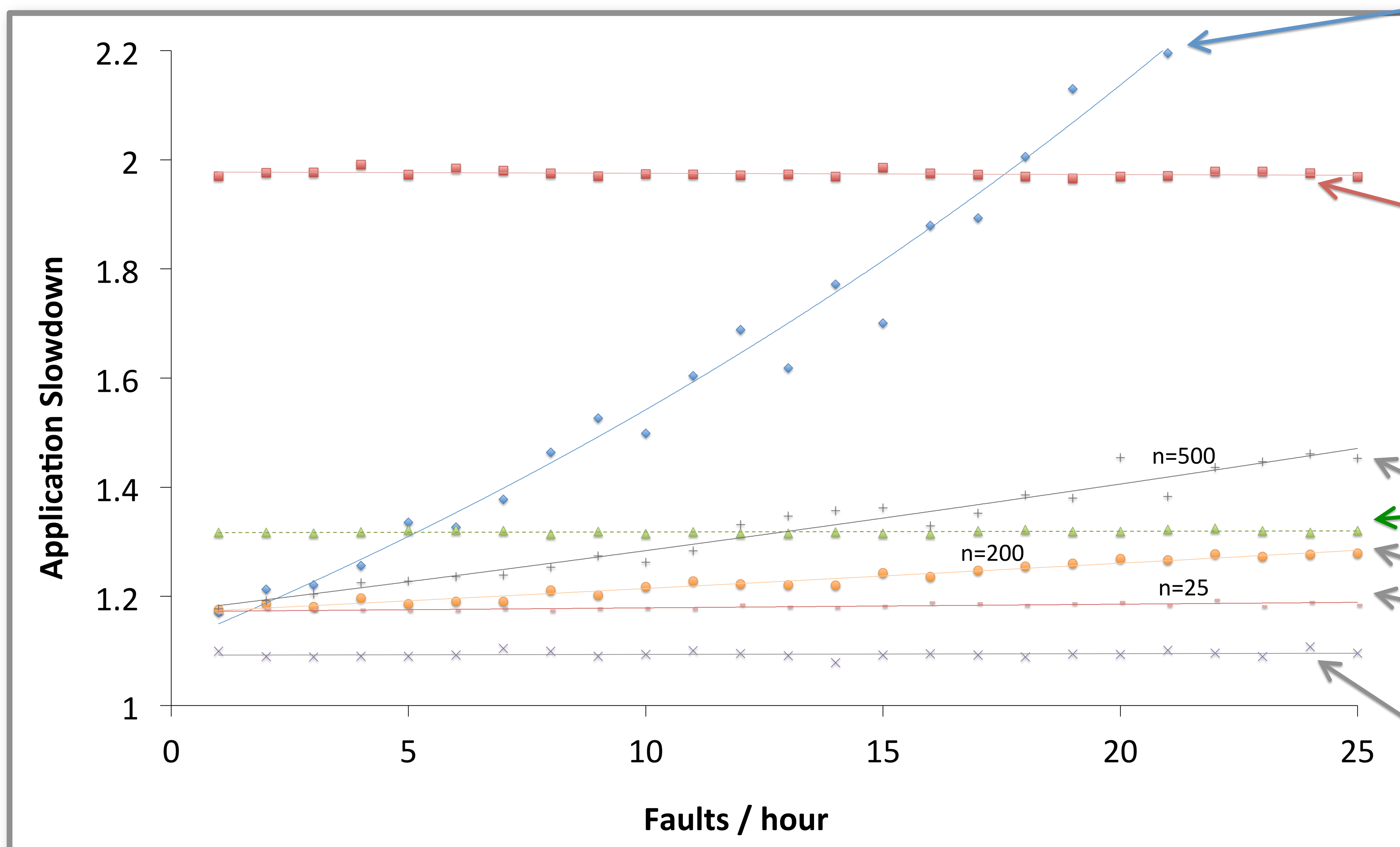


How do we annotate the code?

```
main() {
   /* init code...*/
   while() {
      funct1();
      funct2();
      funct3();
   }
}
```

Re-executing the last *n* iterations of the main loop is optimal

Method 1 performs surprisingly well for low fault rate



Method 1

```
main() {
RETRY {
  while() {
     funct1();
     funct2();
     funct3();
  }
}
}
```

Method 2

```
main() {
RETRY {
  while() {
     RETRY {funct1();}
     RETRY {funct2();}
     RETRY {funct3();}
  }
}
}
```

Method 3

```
main() {
RETRY {
  while() {
    RETRY {
      funct1();
      funct2();
      funct3();
    }
  }
}
}
```

Method 4

```
main() {
RETRY {
  while() {
    RETRY(n) {
      funct1();
      funct2();
      funct3();
    }
  }
}
}
```

```
Idempotent
source code
method (RAJA)
```

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