Resilient X10
Efficient failure-aware programming
Resiliency Spectrum

Node failure is a reality on commodity clusters
- Hardware failure
- Memory errors, leaks, race conditions (including in the kernel)
- Evictions
- Evidence: Popularity of Hadoop

Ignoring failures causes serial MTBF aggregation:
24 hour run, 1000 nodes,
6 month node MTBF
=> under 1% success rate

Transparent checkpointing causes significant overhead.
Resilient X10 Overview

Provide helpful semantics:
• Failure reporting
• Continuing execution on unaffected nodes
• Preservation of synchronization: HBI principle (described later)

Application-level failure recovery, use domain knowledge
• If the computation is approximate: trade accuracy for reliability (e.g. Rinard, ICS06)
• If the computation is repeatable: replay it
• If lost data is unmodified: reload it
• If data is mutated: checkpoint it
• Libraries can hide, abstract, or expose faults (e.g. containment domains)
• Can capture common patterns (e.g. map reduce) via application frameworks

No changes to the language, substantial changes to the runtime implementation
• Use exceptions to report failure
• Existing exception semantics give strong synchronization guarantees
X10 Language Overview (Non-distributed features)

- Java-like language
- Developed ~ 10 years (open source)
- Structs (compound value types)
- Reified Generics

- Activities
  - Lightweight threads
  - Exception propagation
  - Atomic construct

```java
class Test {
    public static def main(args: Rail[String]) {
        finish {
            async {
                Console.OUT.println("1a");
            }
            async {
                Console.OUT.println("1b");
            }
        }
        Console.OUT.println("2");
    }
}
```

Possible interleavings:

- 1a
- 1b
- 2

Or:

- 1b
- 1a
- 2
X10 Language Overview (Distributed Features)

- Scales to 1000s of nodes
- Asynchronous PGAS (APGAS)
  - Heap partitioned into ‘places’
  - Can only dereference locally
- Explicit communication
- Implicit object graph serialization

```scala
class MyClass {
  public static def main(args: Rail[String]): void {
    val c = GlobalRef(new Cell[Long](0));
    finish {
      for (p in Place.places()) {
        async {
          at (p) {
            val v = ...; // non-trivial work
          }
          at (Place.FIRST_PLACE) {
            val cell = c();
            atomic { cell(cell() + v); }
          }
        }
      }
      Console.OUT.println("Cumulative value: " + c());
    }
  }
}
```

```
val x = ...;
val y = ...;
at (p) {
  val tmp = x + y;
}
```

```java
class MyClass {
  public static def main(args: Rail[String]): void {
    val c = GlobalRef(new Cell[Long](0));
    finish {
      for (p in Place.places()) {
        async {
          at (p) {
            val v = ...; // non-trivial work
          }
          at (Place.FIRST_PLACE) {
            val cell = c();
            atomic { cell(cell() + v); }
          }
        }
      }
      Console.OUT.println("Cumulative value: " + c());
    }
  }
}
```
Resilient X10 (Language design)

Sometimes, an arbitrary place may disappear.

Immediate Consequences:
- The heap at that place is lost
- The activities are lost
- Any ‘at’ in progress immediately terminates with `x10.lang.DeadPlaceException`
  (Very similar to `java.lang.VirtualMachineError`)

Lasting Consequences:
Place will never come back alive.
Can no-longer at (dead_place) {...} – get DeadPlaceException thrown.
GlobalRef[T] to objects at that place may still be dangling…
But type system requires use of ‘at’ to access that state.
Code can test if a given Place value is dead, get list of alive places, etc.
Revision of earlier example for failure-reporting X10:

```java
class MyClass {
    public static def main(args:Rail[String]):void {
        val c = GlobalRef[Cell[Int]](new Cell[Int](0));
        finish {
            for (p in Place.places()) {
                async {
                    try {
                        at (p) {
                            val v = ...; // non-trivial work
                            at (Place.FIRST_PLACE) {
                                val cell = c();
                                atomic { cell(cell() + v); } // cell() += v
                            }
                        }
                    } catch (e:DeadPlaceException) {
                        Console.OUT.println(e.place+" died.");
                    }
                }
            }
            // Runs after remote activities terminate
            Console.OUT.println("Cumulative value: "+c());
        }
    }
}
```
Special treatment of place 0

- Activities are rooted at the ‘main’ activity at place zero.
- If place zero dies, everything dies.
- The programmer can assume place 0 is immortal.
  - MTBF of n-node system = MTBF of 1-node system
  - Having an immortal place 0 is good for programmer productivity
    - Can orchestrate at place 0 (e.g. deal work)
    - Can do (trivial) reductions at place 0
    - Divide & conquer expressed naturally
    - Can do final result processing / user interface
- However…
  - Must ensure use of place 0 does not become a bottleneck, at scale
Happens Before Invariance (HBI) Principle

*Failure of a place should not alter the happens before relationship.*

```java
val gr = GlobalRef(new Cell[Int](0));
try {
    finish at (Place(1)) async {
        finish at (Place(0)) async {
            gr()(10);  // A
        }
    }
} catch (e:MultipleExceptions) { }
gr()(3);  // B
assert gr()() != 10;
```

A happens before B, **even if place 1 dies**.

Without this property, avoiding race conditions would be very hard. But guaranteeing it is non-trivial, requires more runtime machinery.
HBI – Subtleties

Relationship between at / finish and orphans

Orphaned activities are *adopted* by the next enclosing *synchronization point*.

\[
\text{at } (\text{Place}(1)) \{ \text{finish async } S \} Q \quad \text{// } S \text{ happens before } Q
\]

\[
\text{finish} \{ \text{at } (\text{Place}(1)) \{ \text{async finish async } S \} Q \} \quad \text{// } S \text{ concurrent with } Q
\]

Exceptions

Adoption does not propagate exceptions:

\[
\text{at } (\text{Place}(1)) \{
\quad \text{try } \{
\quad \quad \text{finish at } (\text{Place}(0)) \text{ async } \{ \text{throw e; } \}
\quad \}
\quad \text{catch } (\text{e:Exception}) \{ \}
\}
\]

// e should never appear here
Implementation: X10 Architectural Overview

Runtime stack:
- `async { ... }
- `finish { ... }
- `at (p) { ... }
- OS threads
- Serialization
- `at (p) async { ... }
- `here
- launching processes

Key:
- Java
- C++
- X10

X10 application
X10 runtime
C++ runtime
Java runtime
JNI wrapper
X10RT (network layer)
PAMI
MPI
Sockets
...
Implementing Resilient X10 (X10RT)

Focus on sockets backend

- We have complete control
- Handle TCP timeouts / connection resets gracefully
- Communicate failures up the stack
- Abort on timeout during start-up phase

Changes to X10RT API:

Simple c++ code to send an asynchronous message and wait for a reply (via X10RT API):

```c
x10rt_send_msg(p, msgid, buf);
while (!got_reply) {
    x10rt_probe();
}
```

becomes

```c
int num_dead = x10rt_ndead();
x10rt_send_msg(p, msgid, buf);
while (!got_reply) {
    int now_dead = x10rt_ndead();
    if (now_dead != num_dead) {
        num_dead = now_dead;
        // account for failure
        break;
    }
    x10rt_probe();
}
```
The implementation reduces ‘at’ to a special case of ‘finish’.

Abstractly, finish is a set of counters

Simplified illustration:

```scala
def finish { val v = new FinishCounters(); ... f.wait(); // may throw MultipleExceptions }
```

```scala
def async { f.begin(...); (); // may communicate ... f.end(...) // may communicate }
```

Counters are used to
- Wait for termination
- Throw DeadPlaceException
3 Possible Finish Implementations

Finish counters need to survive failure of place holding FinishCounters object...

- **Store all finish state at place zero.**
  - Simple
  - Makes use of ‘immortal’ place zero.
  - No problem: If finishes are logically at place zero in the code.
  - Otherwise: Bottle neck at place zero.

- **Store all finish state in ZooKeeper**
  - From Hadoop project
  - External paxos group of processes
  - Lightweight resilient store
  - Still too much overhead (details in paper)

- **Distributed resilient finish.**
  - Finish state is replicated at one other node.
  - Execution aborted if both nodes die.
  - Best all round performance
  - No bottle neck at place zero
Finish Micro-benchmark results
Application – K-Means (Lloyd’s algorithm)

Machine learning / analytics kernel.
Given N (a large number) of points in 4d space (dimensionality arbitrary)
Find the k clusters in 4d space that approximate points’ distribution

• Each cluster’s position is iteratively refined by averaging the position of the set of points for whom that cluster is the closest.
• Very dense computational kernel (assuming large N).
• Embarrassingly parallel, easy to distribute.
• Points data can be larger than single node RAM.
• Points can be split across nodes, partial averages computed at each node and aggregated at place 0.
• Refined clusters then broadcast to all places for next iteration.

Resiliency is achieved via decimation
• The algorithm will still converge to an approximate result if only most of the points are used.
• If a place dies, we simply proceed without its data and resources.
• Error bounds on this technique explored in Rinard06

Performance is within 90% of non-resilient X10
Application – Iterative Sparse Matrix * Dense Vector

Kernel found in a number of algorithms, e.g. GNMF, Page Rank, …
An N*N sparse (0.1%) matrix, G, multiplied by a 1xN dense vector V
Resulting vector used as V in the next iteration.
Matrix block size is 1000x1000, matrix is double precision

G distributed into row blocks. Every place starts with entire V, computes fragment of V’.
Every place communicates fragments of V to place 0 to be aggregated.
New V broadcast from place 0 for next iteration (G is never modified).

Code is memory-bound, amount of actual computation quite low
Problem is the size of the data – does not fit in node.
G is loaded at application start, kept in RAM between iterations.

Resiliency is achieved by replaying lost work:
• Place death triggers other places to take over lost work assignment.
• Places load the extra G blocks they need from disk upon failure

100x faster than Hadoop
Resilient X10 ~ same speed as existing X10
Application – Heat Transfer

Demonstration of a 2D stencil algorithm with simple kernel
An N*N grid of doubles
Stencil function is a simple average of 4 nearest neighbors

Each iteration updates the entire grid.
Dense computational benchmark
Distributed by spatial partitioning of the grid.
Communication of partition outline areas required, each iteration.

Resiliency implemented via checkpointing.
Failure triggers a reassignment of work, and global replay from previous checkpoint.
Checkpoints stored in an in-memory resilient store, implemented in X10

Performance can be controlled by checkpoint frequency.
If no checkpoints, performance is the same as existing X10
Conclusions

Resilient X10

- A novel point in the design space
- Avoid sacrificing performance
- Re-use exception semantics
- HBI principle ensures that transitive synchronization is preserved after node failure
- Ensure no surprises for the programmer

Implemented, tested at scale, released (X10 2.4.1)

- Implemented ‘finish’ 3 ways, microbenchmarked
- Implemented 3 apps that handle failure in different ways
  - K-Means (decimation)
  - Sparse Matrix * Dense Vector (reload & replay)
  - Stencil (checkpointing)
- Apps are extended from non-resilient versions to handle DeadPlaceException
- Performance close to existing X10, but resilient to a few node failures
Papers

- PPoPP 2014 “Resilient X10: Efficient failure-Aware Programming”
- ECOOP 2014 “Semantics of (Resilient) X10”

Future Work

- More applications!
- “Elastic” X10
  - Expand into new hardware
  - Allow new hardware to replace failed hardware
- Tolerate failure of place 0
  - Checkpoint the heap at place 0? Slow place 0, use only for orchestration
  - Or, just don’t have a rooted activity model
Questions?