A Programming Language and Runtime for Distributed Systems with Heterogeneous Fault Characteristics

Sara Achour, Martin Rinard
MIT CSAIL
{sachour, rinard}@csail.mit.edu

1 Abstract

Motivation:

1. Hardware is growing more complex - soft errors are becoming more prevalent. [?]
2. If we were to relax the accuracy constraints we impose on the hardware designers, could we improve performance/lower power consumption/reduce processor area? [if yes: find citations]
3. Existing fault tolerance strategies in distributed systems assume faults are rare, catastrophic - conservatively handle faults by re-running work. Ill equipped to handle frequent, non-catastrophic faults.

System Overview: We present Topaz, a programming language and distributed runtime system that facilitates program development for distributed systems with heterogeneous fault characteristics.

2 Introduction

- Accuracy: How close the computed result is to the correct result.
- Reliability: How often the program returns an accurate result.
- Distributed system with heterogeneous fault characteristics: Machines in the distributed system have varying fault characteristics. We treat these characteristics as unknowns. [Future work: given we know the machine parameters, can we come up with checker/scheduler for a set of unexplored parameters without learning by using learning runs with different parameter values?]

2.1 Background

- Existing work on unreliable computation - Rely
- Existing fault tolerance work in distributed systems. [Look up Byzantine Fault Tolerance]

3 Topaz [Technical Core]

Topaz is a programming language and runtime framework that facilitates program development for distributed systems with heterogeneous fault characteristics. The programmer uses the Topaz taskset control structure to define units of dispatchable computation, or tasks, in the program. Topaz dispatches these tasks to machines, or places, in the system. Topaz tracks per-place failure rates and output error margins for each taskset. The scheduler uses these statistics in conjunction with the output error constraints to distribute tasks across the places. Topaz also accepts a user-defined output checker or generates an auto-checker by learning task outputs. The checkers are used to detect when an error has occurred and reject incorrect outputs.

3.1 Language

Language Constructs: The Topaz language is a C-like language with an additional taskset control structure, which defines a dispatchable set of tasks. The taskset is comprised of optional before, reduce and spec blocks and an obligatory task block. The before block contains code that must be executed before dispatching each task. The task block is preceded by a list of task inputs and outputs; inputs/outputs must be a fixed size primitive array or a primitive value. The reduce, accept and spec blocks have access to the task outputs. The accept/reject blocks define a checker for the task outputs. The spec block defines average error constraints on any of the outputs.

```
float take_sum(float * arr, int n){
  float sum=0;
  taskset take_local_sum (i=0; i<n; i+=CS){
    task
      in(float localarr[CS] = &arr[i])
      out(float lsum)
    {
      int j;
      lsum = 0;
      for(j=0; j < CS; j++){
        lsum += localarr[j];
      }
    }
    reduce{
      sum += lsum;
    }
    accept{
      assert(lsum <= CS && lsum >= 0);
    }
    //tolerate an error of 1 on lsum
    spec (lsum := 1.0)
  }
  return lsum;
}
```

Example: In the above example, the summation of a floating point, positive normalized array is computed by dispatching tasks which compute local array sums, and computing the final sum in the reduction operation. The checker accepts the result if it falls between zero and the chunk size - since the array is positive and normalized the sum cannot fall outside of this range. Alternatively, the programmer can write a reject test or not specify a checker, in which case Topaz will autogenerate a checker for the taskset. The specification enforces an average local sum error be ± 1 for a taskset.

3.2 Runtime System

The runtime system is built on top of OpenMPI [cite]. Worker processes are stateless - they have no knowledge of program state and only need the task message to complete a task. Should I talk about implementation (task packing/unpacking, worker routine, global pointer structures, logging) here?

3.2.1 Task Output Learner

Come up with a provisional checker by running the first couple tasks on the oracle machine, and. A On each reject, dispatch the task to the oracle machine to 'label' the point and use the oracle's output as the task output. If the accept is a close accept (score is low), use the oracle machine to label the accept and use the oracle's output as task output. Track the number of false rejects, true rejects, false accepts, true accepts for the checker - determines checker quality. Uses online SVM learning algorithm. Can augment to use Adaboost if SVM doesn't produce good results.

3.2.2 Task Scheduler

Parameters: (Per Machine) Classifier score (how good classifier is at scoring outputs), % of time it produces bad value, machine score.
Cost function: Try and match specification while maximizing machine score.

3.3 Emulator

The unreliable hardware emulator is an extension of Intel's iAct unreliable hardware tool for Pin [cite]. [Description of iAct deterministic, non-deterministic fault model]. We augmented the emulator to consistently assign hardware fault specifications to processes by process rank and parametrized the included hardware models. With these modifications, we were able to emulate a distributed system with heterogeneous fault characteristics. [This system is ok because it's based on intel architecture.]

4 Results

4.1 Benchmarks

Search: Find parameter values for a system.
Black-Scholes: Predict future stock prices given previous stock info.
Barnes: Planetary motion simulator.
Water: Water motion simulator.
Sum: Matrix sum.

4.2 Output Quality

[Table 1: percent errors on results for Topaz benchmarks and vanilla benchmarks running in emulator]

[Table 2: percent errors on results for Topaz benchmarks with autochecker, without checker, with manual checker]

[Table 3: percent errors on results for Topaz benchmarks with random task distribution, scheduler]

4.3 Power

[Table 4: power savings for each benchmark] Power Approximation: Lift power saving from truffle paper. Argue their unreliability ISA is what iAct implements, than use their power multipliers along with the runtime of the tasks on each place to determine the power savings. It won't be much because their top savings % is 40%, and only a fraction of the computation will be using that.

4.4 Performance

Not sure how to get actual performance numbers. Perusing literature.