GREMLINs
A Tool Infrastructure for System Emulation

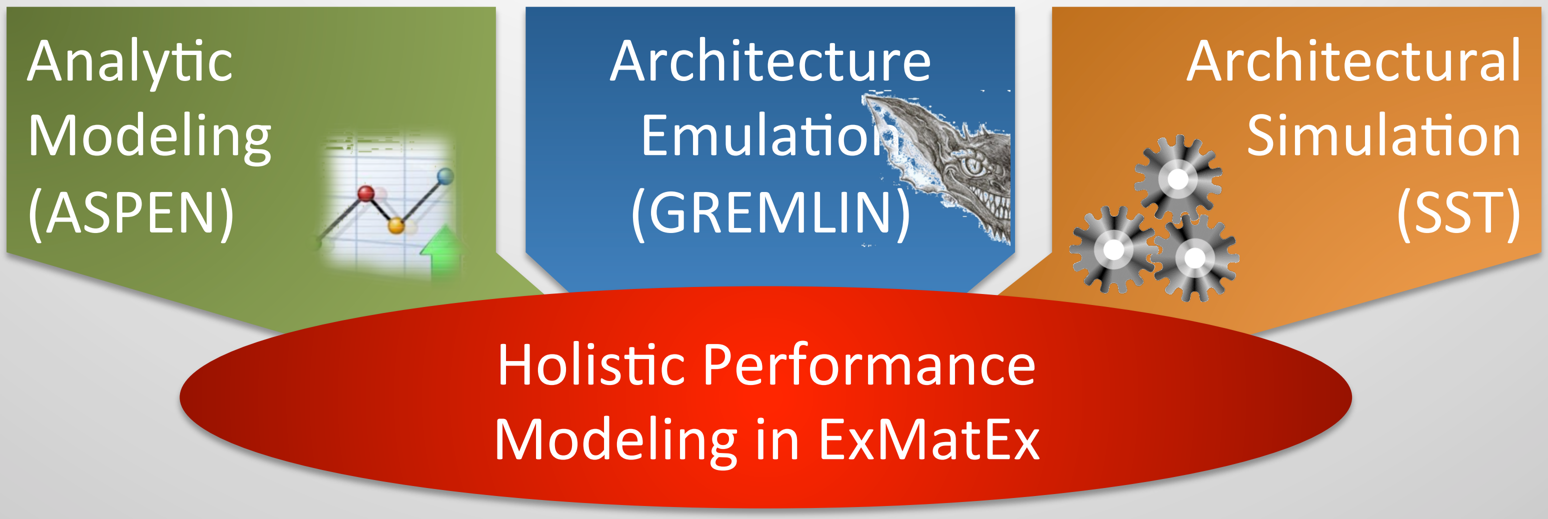
Martin Schulz
Lawrence Livermore National Laboratory
with Barry Rountree, Marc Casas Guix, Greg Bronevetsky, Ignacio Laguna

CScADS Workshop, University of Wisconsin ◆ July 2013

LLNL-PRES-641078

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

Designing a Tool Set for Exascale Co-Design



- **Analytical models provide high-level trends**
  - But don't cover low level details
- **Simulations enable access to architectural details**
  - But are slow and difficult to use with complex codes / validation?
- **Augment with emulation techniques**
  - Run complex codes on real systems

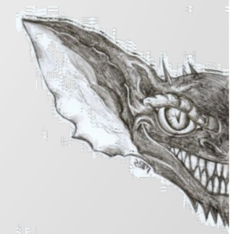
The GREMLIN Idea

- **Can we make a Petascale class machine behave like what we expect Exascale machines to look like?**
  - Exascale machines will be
    - Resource limited (power, memory, network, I/O, …)
    - Have less favorable compute/bandwidth ratios
    - Higher fault rates and lower MTBF rates
- **GREMLINs are a set of techniques to emulate such behavior**
  - Framework to couple range of “bad behaviors”
  - Transparent to system and (mostly) to applications
- **The role in the Co-Design process**
  - Evaluate proxy-apps with GREMLINs and compare to baseline
  - Determine bounds of behaviors proxy apps can tolerate
  - Drive changes in proxy apps to counter-act GREMLINs

Broad Classes of GREMLINs

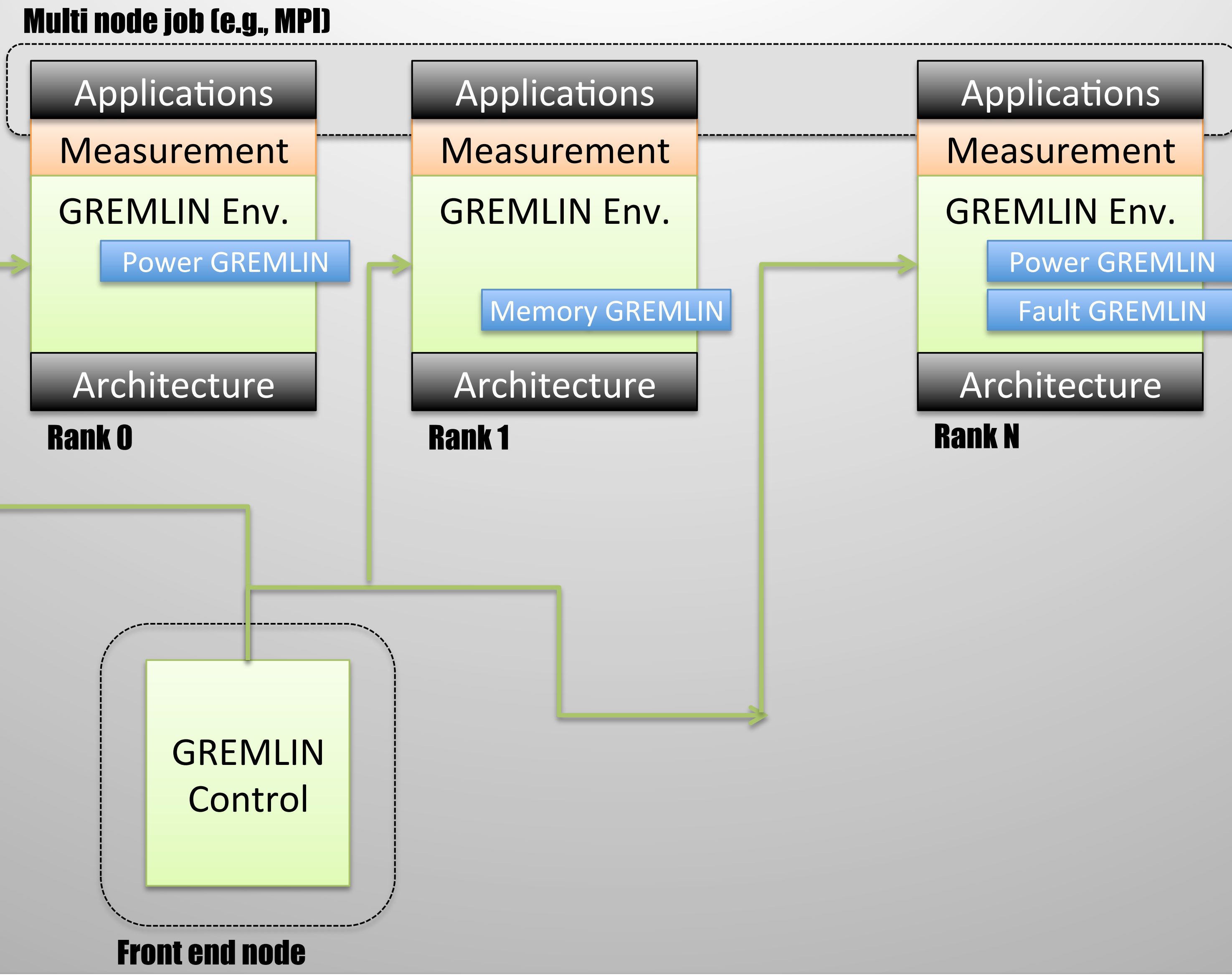
- **Power**
  - Impact of changes in frequency/voltage
  - Impact of limits in available power per machine/rack/node/core
- **Memory**
  - Restrictions in bandwidth
  - Reduction of cache size
  - Limitations of memory size
- **Resiliency**
  - Injection of faults to understand impact of faults
  - Notification of "fake" faults to test recovery
- **Noise**
  - Injection of controlled or random noise events
  - Crosscut summarizing the effects of previous GREMLINs

Implementation Principles



- **Individual GREMLINs are implemented as modules**
  - One effect at a time
  - Orthogonal to each other
  - Each GREMLIN has “knobs” to control behavior
- **Infrastructure to allow loading of GREMLINs**
  - Easy experiment setup using P^nMPI (infrastructure to manage tools)
    - — Enables stacking of PMPI tools
    - — Transparent to applications
    - — Concurrent use of multiple GREMLINs/effects
  - Interactive access to GREMLIN “knobs”
    - — Goal: Python (or similar) driver to influence behavior
    - — Scalable infrastructure (CBTF) for data collection and analysis

Architecture



Needed: Redesign of P^{n}MPI

- **Current design of P^{n}MPI is limited**
  - Static tool stacks
  - Focused on MPI only
- **Enable more dynamic loading options**
  - Load/enable modules on the fly
  - More flexible configurations
  - Separate tool stacks for each process
- **Interceptions of new APIs beyond MPI**
  - How to integrate OMPT?
  - Wrapping library APIs
- **Integration with MPIecho**
  - Cloning of individual ranks to allow concurrent parameter studies

Designing and Deploying a GREMLIN

- **Step 1: Identify target resource**
  - Which resource is supposed to be reduced/controlled/injected into?
- **Step 2: Find mechanism to control/restrict resource**
  - Hardware mechanisms (e.g., RAPL)
  - Direct software techniques (e.g., injection)
  - Indirect software techniques (resource stealing)
- **Step 3: Measurement techniques**
  - Application performance metrics
  - Co-execution with tools
- **Step 4: Mitigation mechanisms**
  - How can the effect of a GREMLLIN can be mitigated?
  - Design of new runtime systems (e.g., Adagio)
  - Fault resilience techniques to react to fault injections

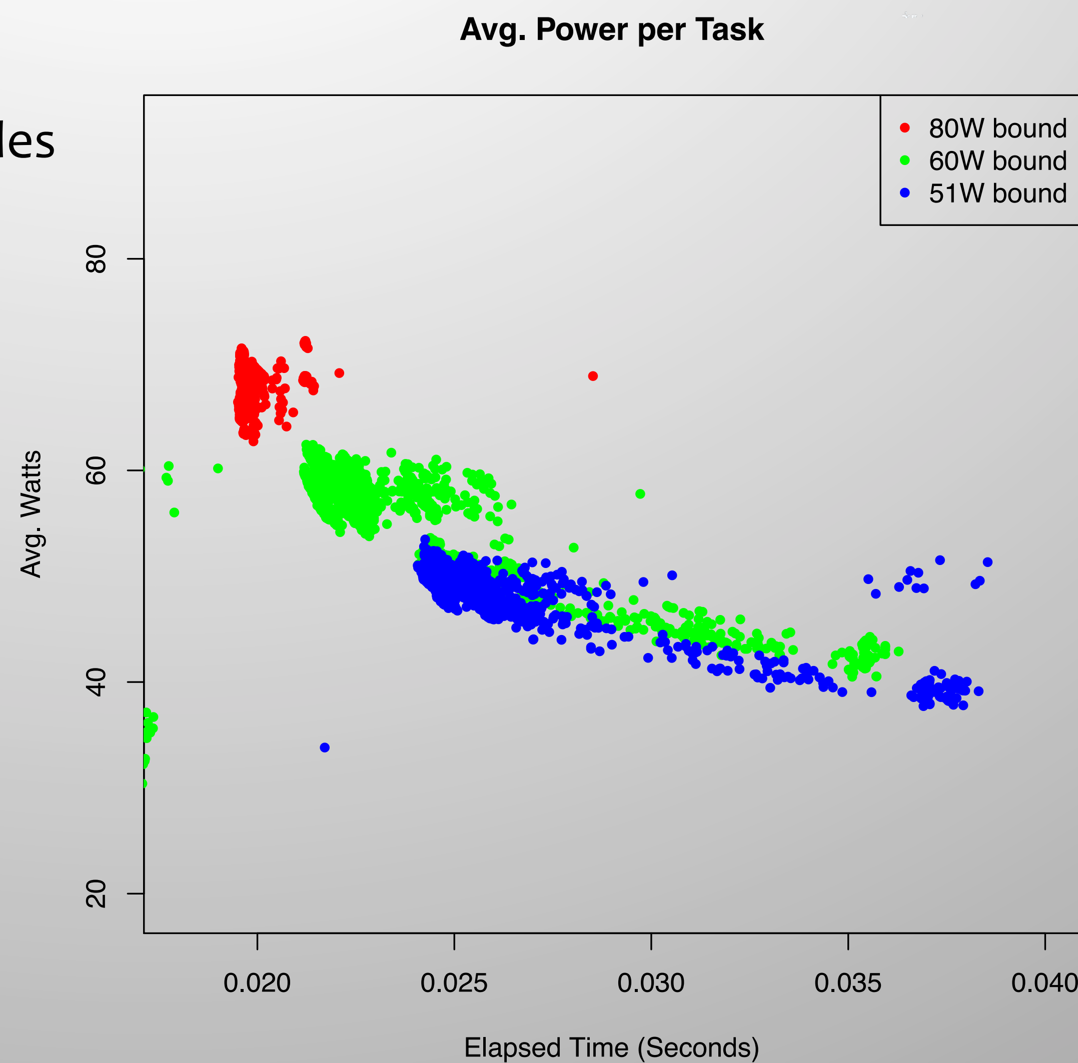
Power GREMLINs

- **Investigate impact of constrained power on applications**
  - Changes in frequency/voltage to save power
  - Overall power caps imposed by machine limits (per system/rack/…)
  - Local power caps for overprovisioned chips with dark silicon

- **Implementation**
  - Access to power measurements on Intel Sandy Bridge and BG/Q
  - Changes of power caps on Intel Sandy Bridge using RAPL
    - — Production machine with the ability to do large scale runs
    - — Emulation of over provisioned systems
  - GREMLIN functionality mainly limited to initialization

Example: CoMD under Multiple Power Bounds

- **MD proxy app**
  - 128 MPI ranks over 8 nodes
  - Dual socket 8-core
  - RAPL measurements (avg. package power)
- **Observations**
  - Lower cap leads to lower performance
  - Lower cap leads to more variation
- **Power capping can lead to load imbalance**



Power Analysis with GREMLINS

- **Co-Design questions**
  - What is the optimal configuration for a given power budget?
  - How will we deal with over-provisioned systems?
  - Which parts of a code a most sensitive to power caps?
  - How do automatic techniques interfere with the software stack?
  - How to direct power where it is needed?
- **Mitigation options**
  - Critical path based analysis and power control
  - Global information to steer local adaptations
- **Requirements**
  - Precise, predictive power models
  - Flexible access to power control mechanisms in hardware

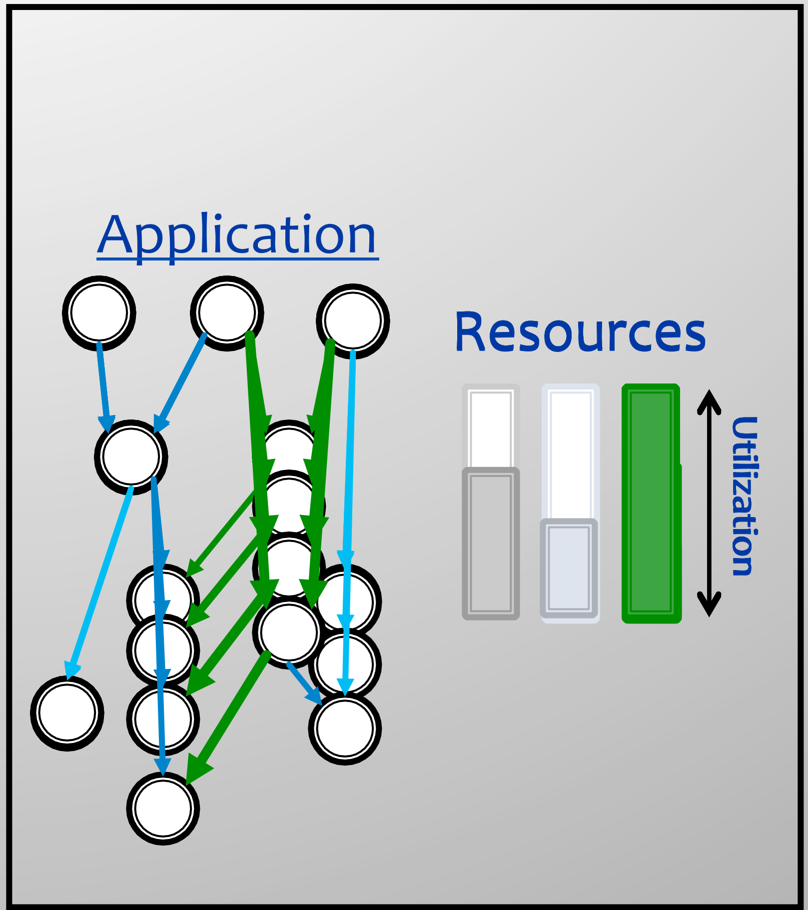
Memory GREMLINs

- **Investigation of limitations in the memory system**
  - Identification of non scalable memory requirements
  - Investigation of “breaking points” for apps wrt. bandwidth/caching
- **Implementation (targeting size)**
  - Wrappers of all memory allocation routines
  - Allocate multiple times the size of the request (or tracking/extrapolating)
- **Implementation (targeting bandwidth/caches)**
  - Resource stealing (more on next slide)
- **Mitigation mechanisms**
  - Locality optimizations (app)
  - Communication avoiding algorithms (app)
  - Scheduling optimizations (system)

Measurements Using Resource Stealing

- **Overload resource**
  - Observe impact on application
  - Study breaking point

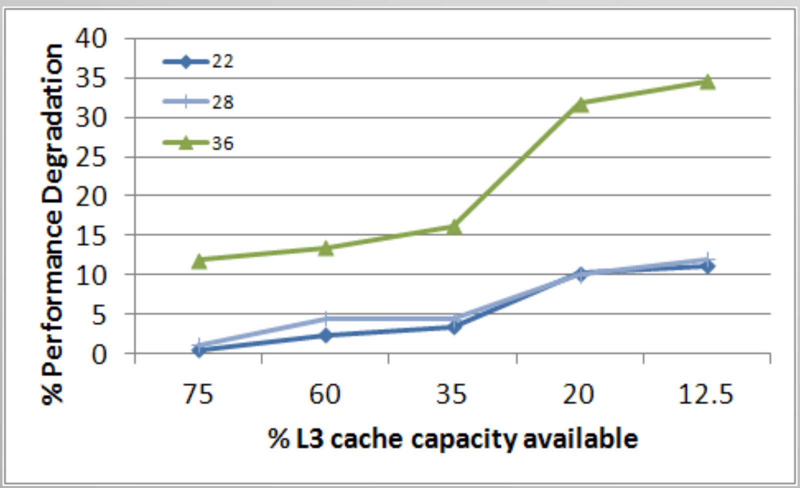
- **Implementation: Interference workload**
  - Additional threads adding bandwidth to a bus
  - Separate thread utilizing a predefined part of the cache



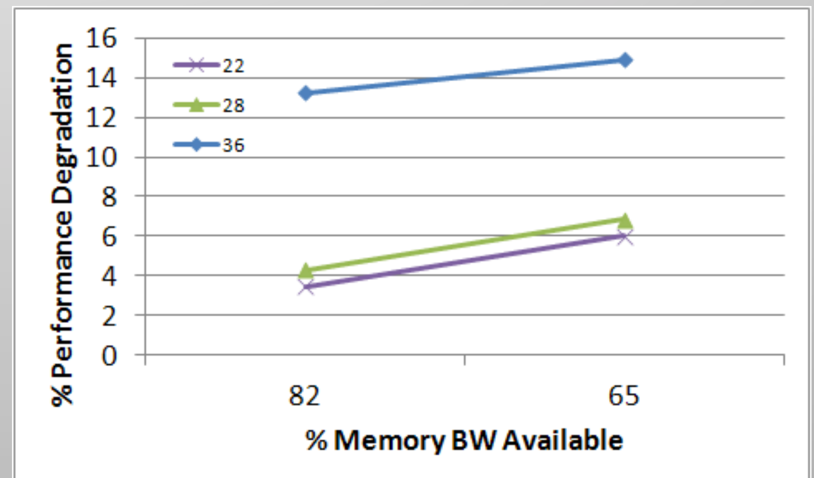
Parallel Application Study: Lulesh

- **Lulesh: Shock Hydrodynamics proxy app**
  - 64 MPI ranks, one task per socket / two per node
  - Cache interference: random touches in predefined memory region
  - Bandwidth interference: walk large buffer

Cache Capacity Analysis



Bandwidth Analysis



GREMLINs (Resiliency)

- **Investigation of reduced reliability**
  - What can applications tolerate as is?
  - What resiliency techniques are needed if faults go beyond that?
  - At what point does a system become infeasible?
- **Implementation (targeting actual faults)**
  - Fault injection with various mechanism
    - Binary rewriting (DynInst), LLVM, dynamic rewriting (PIN), …
  - Vulnerability studies
  - Recovery testing
- **Implementation (targeting “fake” faults)**
  - Injection by invoking correction handler inside the application
  - Evaluate overhead and feasibility of mitigation mechanisms

Early Study on Application-Level Recovery

- **Simple *retry* code blocks**
  - Programmer annotates (or protect) code block
  - If error occurs, code block is re-executed
  - Retry until block terminates without errors

Original code

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

Annotated code

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

- **Fault model**
  - Hardware errors detected by hardware
  - Notification through OS that triggers RETRY block
  - Triggered by a GREMLIN ("fake" fault)

Try/Catch Methods in LULESH

LULESH

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

Method 1
MAIN_FUNC_ONLY

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

Method 2
CORE_FUNCTIONS

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

Method 3
CORE_LOOP

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

Method 4
i_ITERATIONS_BACK

```
main() {
TRY {
   while() {
      TRY(25) {
      funct1();
      funct2();
      funct3();
      }
   }
}
}
```

```
main() {
TRY {
   while() {
      TRY(100) {
      funct1();
      funct2();
      funct3();
      }
   }
}
}
```

```
main() {
TRY {
   while() {
      TRY(200) {
      funct1();
      funct2();
      funct3();
      }
   }
}
}
```

```
main() {
TRY {
   while() {
      TRY(500) {
      funct1();
      funct2();
      funct3();
      }
   }
}
}
```

Normal LULESH Run

```
lagunaperalt1 — ssh — 89×31
[[redacted]@sierra0 demo]$ ./run_experiment.py ./lulesh

Entering main loop
.........................................................................................
.........................................................................................
.........................................................................................
.........................................................................................
.................................................................

Iteration count = 511
Final Origin Energy = 1.254873e+06
Testing Plane 0 of EnergyArray:
        MaxAbsDiff = 1.746230e-10
        TotalAbsDiff = 2.194418e-09
        MaxRelDiff = 2.966367e-13

RUNTIME (sec): 65.926990
[[redacted]@sierra0 demo]$
```

Each iteration prints "."

Output and runtime is printed at the end

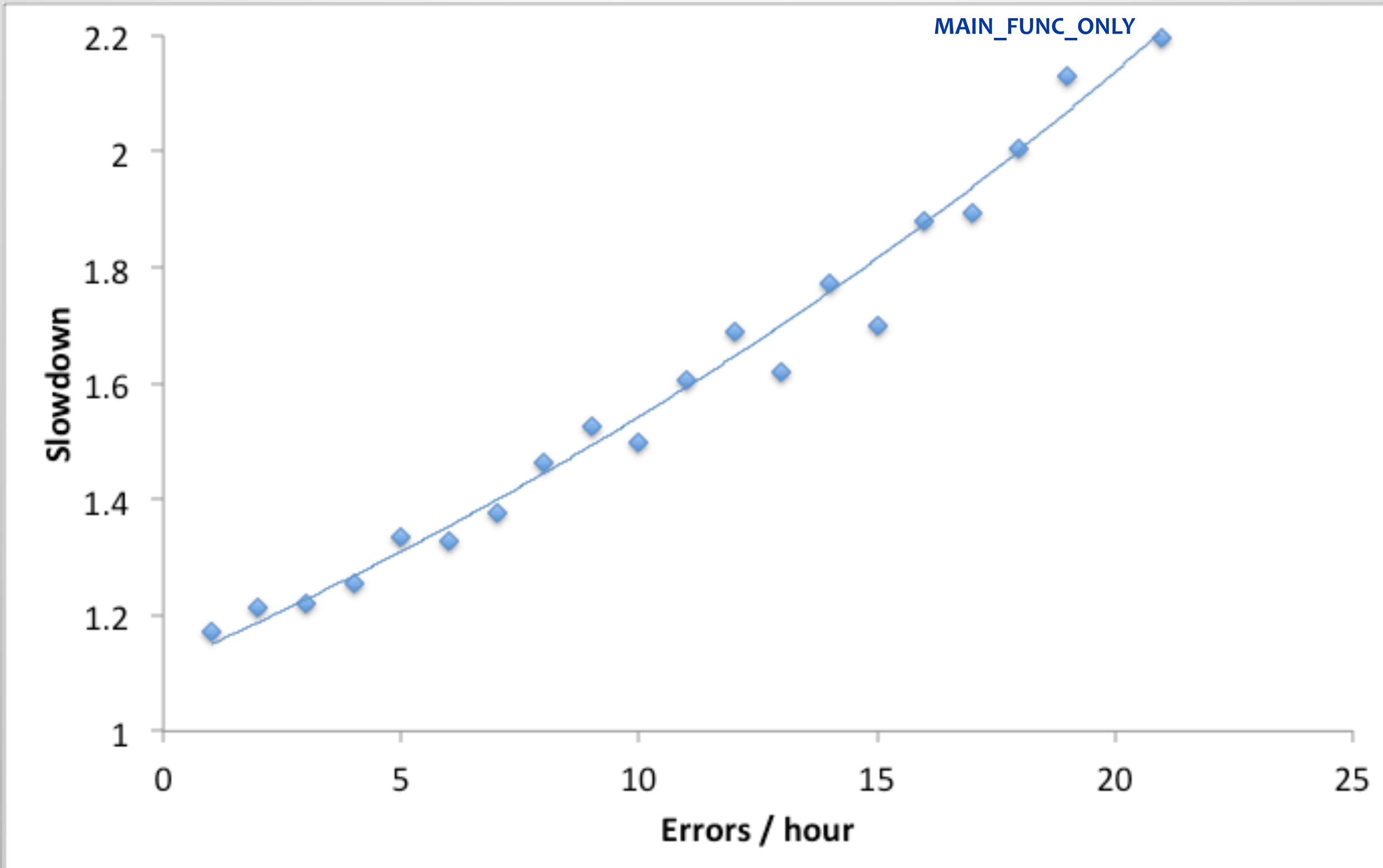
Error rate

Faulty LULESH Run

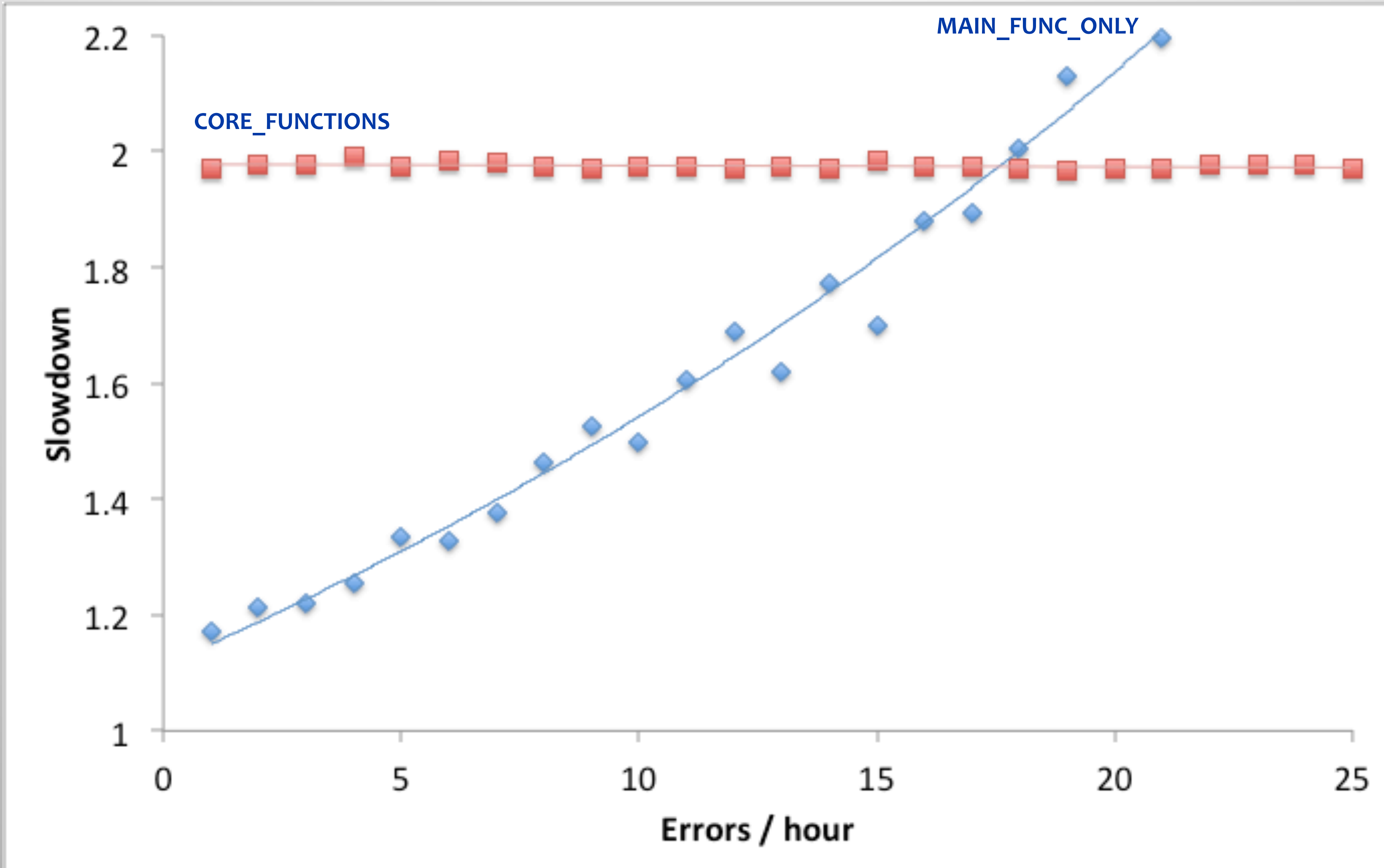
```
lagunaperalt1 — ssh — 85×29
[[redacted]@sierra0 demo]$ ./run_experiment.py -e 500 ./lulesh_core_functions
-----------------------------------------------------
Injecting at 500 errors/hour
-----------------------------------------------------


Entering main loop
.........................................................................X.........
.....................................................................X..............
...............................................................................X....
.....................................................................................
...........................X......................X..X..............................
...........................................................................X...X
.....X....

Iteration count = 511
Final Origin Energy = 1.254873e+06
Testing Plane 0 of EnergyArray:
        MaxAbsDiff = 1.746230e-10
        TotalAbsDiff = 2.194418e-09
        MaxRelDiff = 2.966367e-13

RUNTIME (sec): 100.750642
[[redacted]@sierra0 demo]$
```

When error occurs, it prints "X"

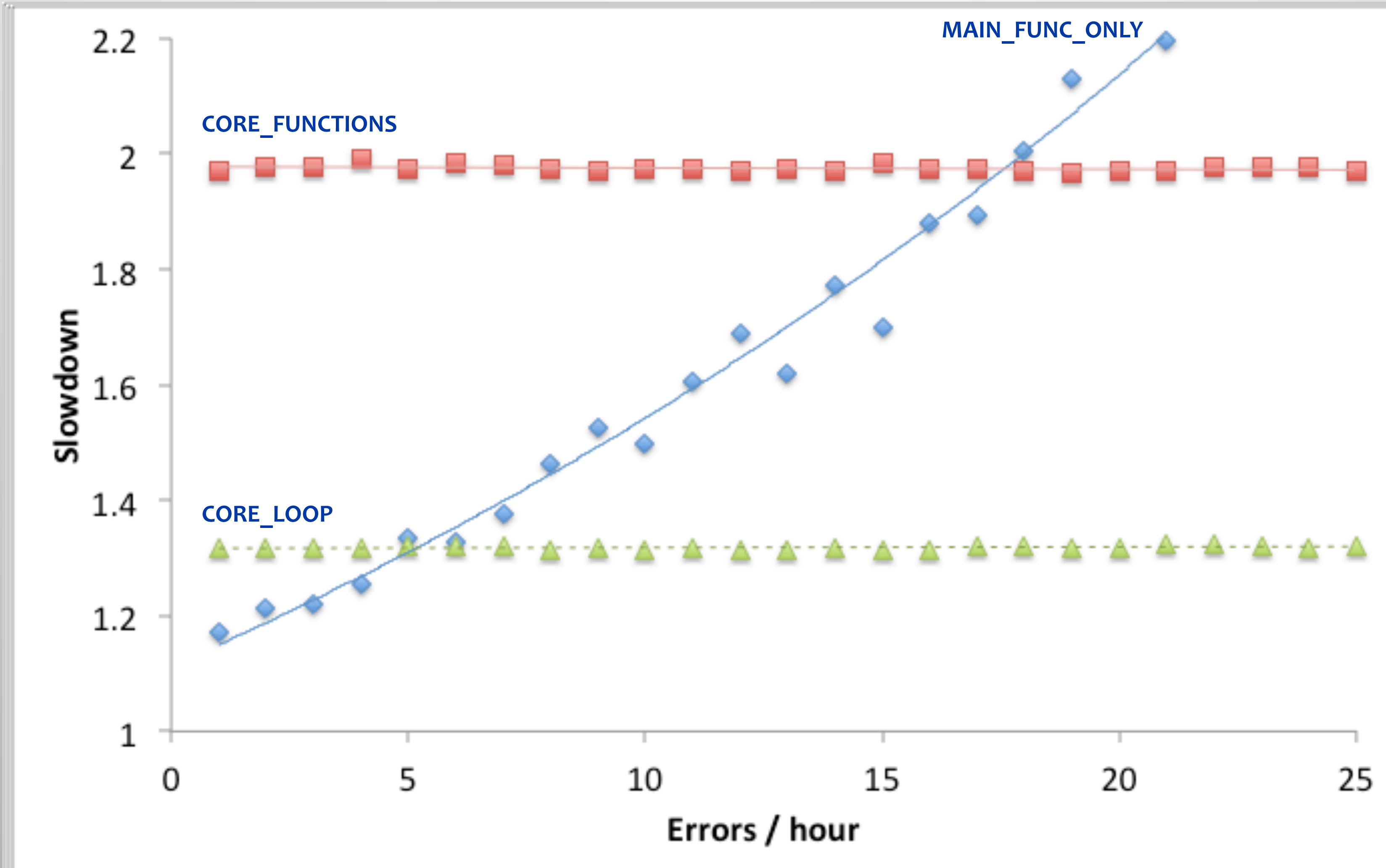
Experimental Results



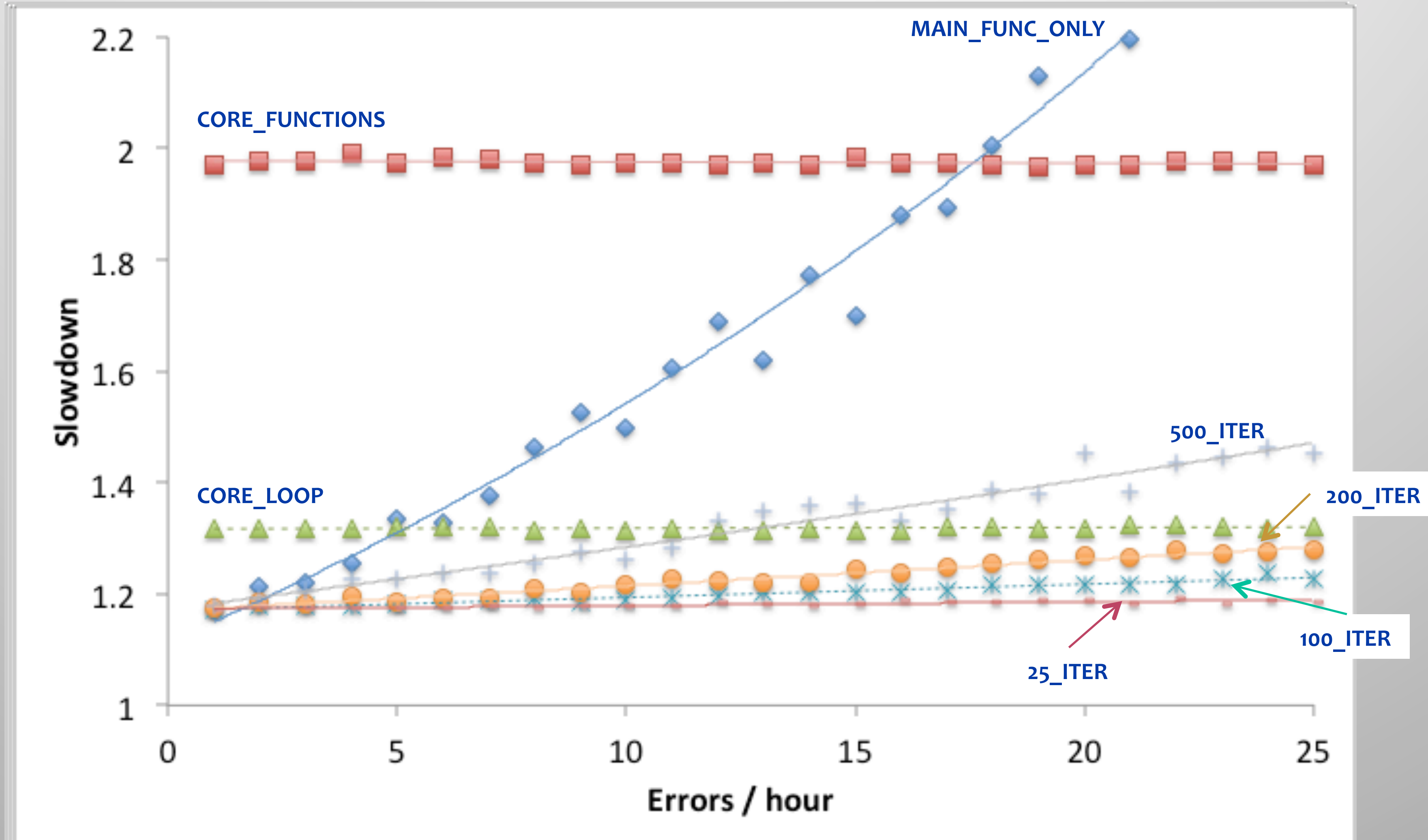
Experimental Results



Experimental Results



Experimental Results



Co-Design Questions for Resilience

- **Fault model, injection, and detection**
  - Integration of injections techniques into GREMLINs
  - Study of error models
  - HW/OS/RT APIs for reporting detected errors
- **Mitigation mechanisms**
  - APIs for applications to expose vulnerable state
  - Fault resilient algorithms
  - Techniques inside the application to recover from faults
    - — Code redundancy
    - — Data reconstruction
  - Investigate fault tolerant MPI proposals
  - Options for direct support in runtime systems

Conclusions and Future Plans

- **Using emulation to support the co-design process**
  - Ability to execute full codes on real machines at scale
  - GREMLIN approach imposes constraints to emulate future architectures
- **GREMLINs can cover many aspects of future systems**
  - Power constraints and their impact
  - Constraint in memory resources
  - Impact of faults and recovery techniques
- **Future work**
  - New GREMLIN/emulation techniques in hardware and software
  - Ensemble of GREMLINs to enable large parameter studies more quickly
  - Integration of GREMLINs into new programming models
  - Integration with new scale bridging MPMD environments
  - Planned for 9/13: release of the GREMLINs