FlipIt: An LLVM Based Fault Injector for HPC

7th Workshop on Resiliency in High Performance Computing (Resilience) in Clusters, Clouds, and Grids

Jon Calhoun, Luke Olson, and Marc Snir

University of Illinois at Urbana-Champaign

25 August 2014

Classification

Faults lead to errors, and can be classified into two groups:

Hard

- Faults that are reproducible (non-transient)–e.g. the inability to communicate with a node that is offline
- Leads to application crashes
- Recover using checkpoint-restart

Soft

- Faults where activation is not systematically reproducible (transient)–e.g. a bit-flip caused by a charged particle
- Leads to a silent data corruption (SDC)
- We can recover *if* we know they occurred

Bit-flips Happen

- Bit-flips can occur by alpha particles or neutrons interacting with silicon based transistors
- Experienced one a daily basis on large scale HPC systems
- BlueGene/L at the LLNL produced an L1 cache bit-flip every 3-4 hours

Fault Model

- Memories are protected by error correcting codes (ECC) and chipkill; therefore, lets not make this our primary focus
- Faults arise in processor computations and manifest as register perturbations (single bit-flip)

What is LLVM?

- The LLVM Project is a collection of modular and reusable Static Single Assignment (SSA)-based compiler and toolchain technologies developed at the University of Illinois
- LLVM is used by a wide variety of commercial and open source projects as well as academic research

Fault Injector

- To enable bit-flips as single bit permutations, our fault injector extends the LLVM fault injector KULFI with the following additions:
  - Support for complex pointer types
  - Ability to work with multiple source files simultaneously
  - User customized fault distribution and event logger
  - Support for a larger subset of the LLVM instructions
  - MPI rank aware

- Fault sites are enumerated at compile time, but activation is done at runtime (dynamic injections)

Compiler Pass

```
define i32 @add(i32 %a, i32 %b) #0 {
entry:
    %add = add nsw i32 %a, %b
    ret i32 %add
}
```

(a) Original LLVM IR.

```
define i32 @add(i32 %a, i32 %b) #0 {
entry:
    %add = add nsw i32 %a, %b
    %crptAdd = call i32 @corrupt(i32 0, i32 1,
                 double 0.01, i32 3, i32 %add)
    ret i32 %crptAdd
}
```

(b) Transformed LLVM IR.

Figure : Code Transformation to Inject Faults.

Corrupt function's arguments: fault site index, boolean for one injection per active rank, injection's probability of experiencing a fault, in which byte to flip the bit, and the data to corrupt

Injection Logic

Algorithm 1: Generic corrupt logic.

Input: $siteProb$: Probability that this instruction is faulty
$siteIndex$: Unique index of this fault site
$data$: Value eligible for corruption

Result: Data unmodified(no injection), or data with a single bit-flip(injection)

```
1 if ¬ shouldInject(injectorOn, siteProb) then
2     return data;
3 else
4     bitPosition ← random bit position in targeted byte;
5     logInjection(siteIndex, bitPosition);
6     data_corrupt ← data ⊕ (0x1 ≪ bitPosition);
7     return data_corrupt;
```

Algorithm 2: Basic `shouldInject` logic.

Input: $siteProb$: Probability that this instruction is faulty
$injectorOn$: Boolean signifying if injector is on
$rankInject$: Boolean signifying if rank is faulty

Result: Boolean signifying if an injection will occur

```
1 P ← probability();
2 if injectorOn and siteProb > P and rankInject = TRUE then
3     return TRUE;
4 else
5     return FALSE;
```

Source Code Modifications

Source Code

- Minimal three line modification to `main` that includes a header file, initializes, and finalizes the fault injector
- Other calls to FlipIt functions can be added to allow more user control

```
#include "/path/to/fault/lib/corrupt.h"
#include <mpi.h>
int main(int argc, char** argv)
{
    MPI_Init(&argc, &argv);
    int id; int seed = 71;
    MPI_Comm_rank(MPI_COMM_WORLD, &id);
    FLIPIT_Init(id, argc, argv, seed);
    foo();
    FLIPIT_Finalize(NULL);
    MPI_Finalize();
}
```

Makefile Modifications

- You can do it the long way by adding the following commands for each source target:
  1. Compile source code to LLVM IR
  2. Run compiler pass over the LLVM IR specifying several arguments
  3. Compile to object code
- To make this process easier and less error prone we provide a script which wraps this process. To use it replace the compiler in the makefile with the script
- To enable successful compilation of MPI application, the compiler command wrapped by `mpicc` is stored inside a config file used by our wrapper to communicate information to the compiler pass

What does the compiler pass generate?

When the compiler pass is operating on the LLVM IR it is enumerating all possible fault sites inside the targeted functions and recording information about the site inside a fault site log

This injection log contains:

- Unique fault site number
- Type of injection based upon data type or usage
  - **pointer** - refers to all calculations directly related to use of a pointers (loads, stores, and address calculation)
  - **control** - refers to all calculations of branching and control flow (comparisons for branches and modification of loop control variables)
  - **arithmetic** - refers to mathematical operations
- If the instruction's result or operand is corrupted
- Source line number

User Control

To enhance user control over fault injection we provide:

- The ability to set a custom probability calculation function
- The ability to define custom probabilities for each instruction type
- The ability to define a custom logging function that will be called on every injection
- The ability to select a subset of MPI ranks for injection
- The ability to change active MPI ranks and turn off/on injector state
- The ability to select a subset of fault sites without need for recompilation

Experiments on Hypre

- We compile sections of Hypre with our fault injector to look at SDCs that arise during the solving of a linear system
- To solve the linear system we use Algebraic Multigrid (AMG) with one iteration of Jacobi relaxation for smoothing
- The problem is a 2D Laplacian with zero on the boundaries
- Profiling Hypre allows us to determine the call stack inside `HYPRE_BoomerAMGSolve`, and we select all these functions for injection
- We test on Blue Waters using 16 processes per node

Scalability on Hypre

Each execution of Hypre has the fault injector on, but the the probability of injecting a fault is set to 0 for all instructions

As we increase the number of MPI processes the overhead of the fault injector decreases

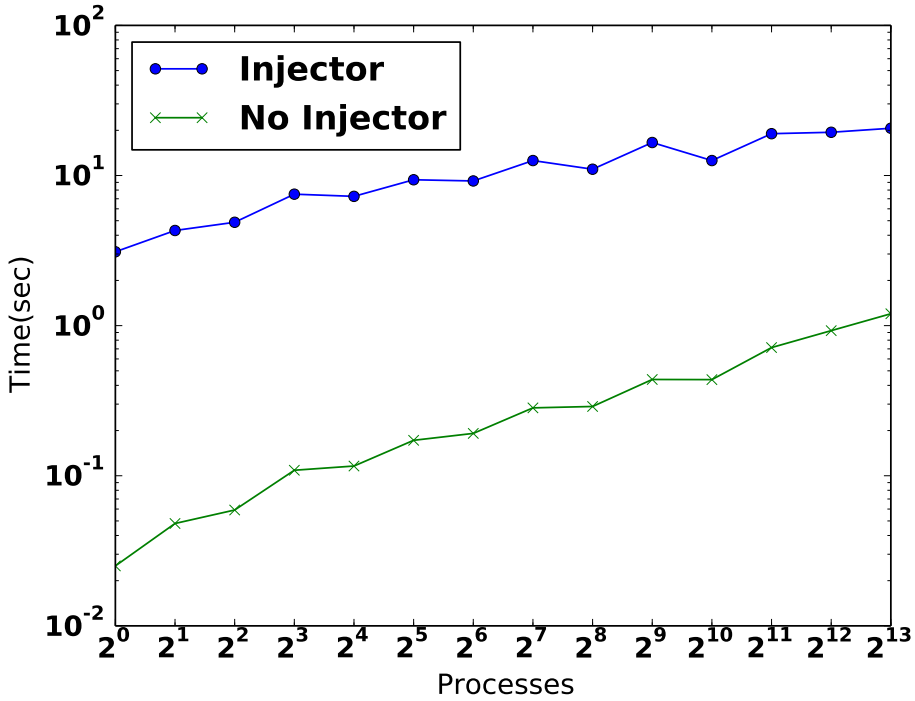

Figure : Weak scaling of Hypre with approximately 16,384 unknowns per process

Selective Injection in Hypre

We selectively flip a bit inside the calculation of the first element of the residual vector on the finest level just before restriction

The name of the trend indicates which bit is flipped in the 64-bit floating point number

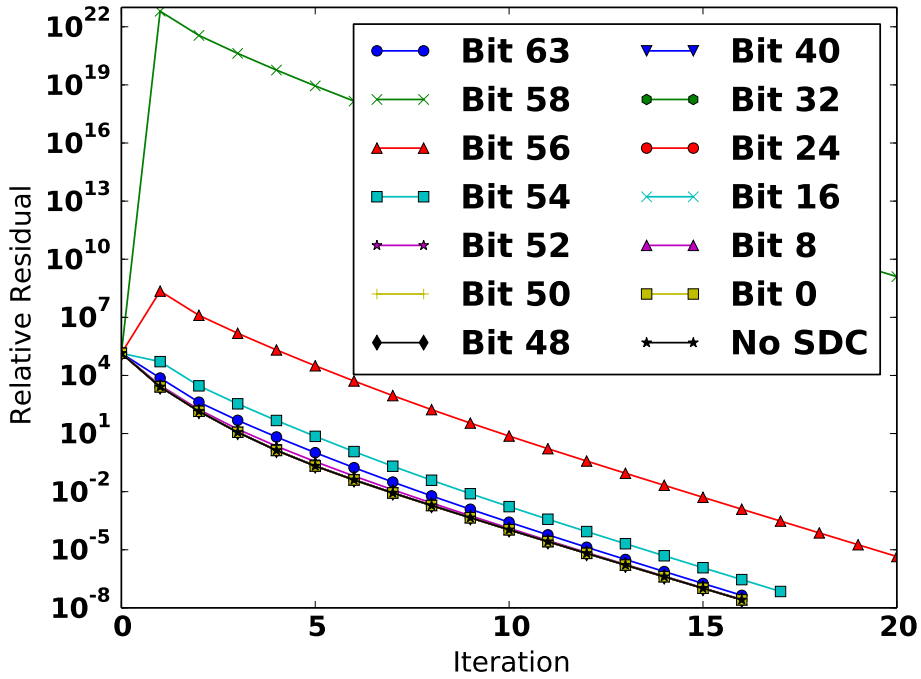

Figure : Selective injection in residual calculation on process 0 of 8 processes with approximately 16,384 unknowns per process.

Effects of Different Injection Types in Hypre

Table : Results of injecting a single fault of a certain type as classified by FlipIt

	Pointer	Control	Arithmetic	All
Crash	41	29	21	29
More Iterations	6	0	6	4
Same Iterations	53	71	73	67

- Injection into pointers has a corresponding increase in the percent of runs that crash
- Injection into the mathematics of AMG increases the percent of runs that require a higher number of iterations required to converge
- Injection into control yields a small increase in the percent of runs that crash due taking incorrect paths and incorrect indexing

Conclusions

- We are headed toward a faulty future, and need to develop and test methods to detect silent errors
- Our fault injector FlipIt provides a high degree of customizability can be used to test these methods

Acknowledgments

- This work was sponsored by the United States Air Force Office of Scientific Research under grant FA9550-12-1-0478

- This research is part of the Blue Waters sustained-petascale computing project, which is supported by the National Science Foundation (awards OCI-0725070 and ACI-1238993) and the state of Illinois. Blue Waters is a joint effort of the University of Illinois at Urbana-Champaign and its National Center for Supercomputing Applications.

Thank you

Any questions?

FlipIt is available on github:
https://github.com/aperson40/FlipIt