ECE 6930-004
HPC Fault Tolerance
Homework 3
Due: 5:00 PM 9 November 2018

This homework will explore soft errors and their impact on running applications. This homework uses the LLVM based fault injector FlipIt to inject bit-flip errors into a running application and analyzes the fault injection campaign with the fault analysis tool FaultSight. **You will get segfaults, but that is okay. Segfaults means its working!**

Part 1: Installing FlipIt (5 points)

Follow the instructions at the link below to install FlipIt:

https://github.com/joncalhoun40/FlipIt

When you are instructed to download LLVM and Clang version 3.5.2 you just need the "Pre-built Binaries" tar file for "Clang for Ubuntu 14.04 Linux" Under 3.5.2. This tar file will give you Clang and LLVM. After finishing the setup, load the gcc/4.8.1 module on Palmetto

```
module load gcc/4.8.1
```

Trying to run some of the examples to verify that FlipIt is working correctly. If you have any issue with running the examples in the examples directory, let Dr. Calhoun know and he can help debug your installation.

Show me run output from the matmul example.

Part 2: Compile HPCCG (5 points)

Download the mini-app HPCCG:

https://mantevo.org/download.php

Modify the `Makefile` variables to the following:

```
CXX = mpicxx

LINKER = mpicxx

USE_MPI = -DUSING_MPI

CPP_OPT_FLAGS = -O1 -g
```

If not already included in your environment issues the command to load the mpich module.

```
module load mpich
```

The mpich module contains an implementation of the message passing interface (MPI). HPCCG uses MPI to exchange messages amongst the parallel processes.

Compile and run the mini-app using `mpicxx` and `mpirun` to verify that it works correctly.

```
mpirun -n 8 ./test_HPCCG 32 32 32
```

Note you will need to use an interactive job or compose a job submission script and submit it with `qsub`. Information on running jobs on Palmetto is found here.

Show me the run output from an execution of HPCCG with the aforementioned command.

Part 3: Instrument HPCCG for fault injection (5 points)

Instrument the HPCCG application for fault injection with FlipIt by inserting the appropriate FlipIt API calls into the file `main.cpp`. Look at the example `examples/mpi/jacobi/` for a reference. You will need to initialize Flipit and finalize FlipIt. Determine the "seed" argument for `FLIPIT_Init()` by grabing the last command line argument and converting it to an integer.

In addition, turn off FlipIt after initialization and turn in on just before the function call to `HPCCG()` with a call to `FLIPIT_SetInjector()` with the correct argument `FLIPIT_OFF` or `FLIPIT_ON`.

Add the following to `main.cpp` just before the definition of `main()` to record what signals (if any) the program raises due to the injected fault.

```
#include "FlipIt/corrupt/corrupt.h"
#include <signal.h>

void sigHandler(int sig)
{
    printf("Receiving Sig %d\n", sig);
    unsigned long long t = FLIPIT_GetExecutedInstructionCount();
    int r;
    MPI_Comm_rank(MPI_COMM_WORLD, &r);
    printf("Rank %d: Total Insts executed = %llu\n", r, 
FLIPIT_GetExecutedInstructionCount());
    exit(sig);
}
```

This function defines a custom signal handler that will be called whenever a signal is raised for our program. To use our custom signal handler when a segmentation fault or bus error occurs, add the following code as the first lines of `main()`:

```
if (signal (SIGSEGV, sigHandler) == SIG_ERR)
        printf("Error setting segfault handler...\n");
    if (signal (SIGBUS, sigHandler) == SIG_ERR)
        printf("Error setting bus error handler...\n");
```

Modify the if conditional that checks to see if `argc` is 4 to always be true. I.e., the following code is always executed:

```
nx = atoi(argv[1]);
ny = atoi(argv[2]);
nz = atoi(argv[3]);
generate_matrix(nx, ny, nz, &A, &x, &b, &xexact);
```

Set the maximal number of iterations, `max_iter`, to 450 and the solver tolerance, `tolerance`, to 1e-10.

Copy the `config.py`, and `HPCCG.config` files from Canvas to the HPCCG directory. Modify the `Makefile` to use the `flipit-cc` compiler from `FlipIt/scripts`, and then run

```
make clean; make.
```

Using an interactive job or a single job submission to verify that faults are being injected.

```
mpirun -n 8 ./test_HPCCG 32 32 32 --stateFile /path/to/FlipIt//.HPCCG --
numberFaulty 1 --faulty 2 29634
```

Show me the output of this fault injection trial. If you do not see a fault injection message something has gone wrong.

Part 4: Conduct a fault injection campaign (5 points)

The file `HPCCG.cpp` contains the function HPCCG that implements a parallel version of the iterative linear solver Conjugate Gradients (CG). CG solve a system of linear equations in the form $Ax = b$. This function contains a loop that iterates for set number of iterations to refine an initial guess of the true solution of `x` at each iteration or until the approximation is deemed sufficiently accurate based on a predefined `tolerance`.

Add code to `HPCCG.cpp` set the `print_freq` to 1 such that the residual norm is printed each iteration. In addition, add code to verify the norm of residual, `normr`, is less than the previous iteration's residual. If the norm of the residual is larger than the previous iteration's, output the following message to the terminal screen.

```
printf("DETECTED SILENT DATA CORRUPTION!!!\n");
```

Use the script `runner.py` to conduct a fault injection campaign of 1500 trials in which each trial experiences a single bit-flip injection on a random MPI process at some point during the linear solve phase.

```
python runner.py 1500 29634
```

The fault injection campaign will be used later during the analysis phase. **Do not include your fault injection trial data in your submission.**

Part 5: Installing FaultSight (5 points)

Follow the instructions at the link below to install FaultSight:

https://github.com/einarhorn/faultsight/

You will need to use the `analysis_config.py` and the updated files for FaultSight from Canvas when reading the injection data. You will need to set directory paths accordingly in `analysis_config.py. When building the FaultSight database it is expected that all the test_* directories are in the same folder.`

Take a screen shot of FaultSight's main page.

Part 6: Fault Injection Analysis with FaultSight (40 points)

Using FaultSight explore the fault injection campaign from Part 4. If you logged extra information, you can define custom parsing functions in `custom.py` to extend the tables, extract your logged information, and insert it into the database. Briefly summarize the results of the fault injection campaign and answer the following questions:

- What are the functions that faults are injected into?
- What is the percentage of fault injection trials that have detection?
- What are the bit locations that have detection?
- What is the percent of trails that segfaulted?
- What are the bit locations that generate a segfault? Why do you think these bits cause a segfault?
- What are the bit locations in `ddot` where we inject faults?
- What line of code suffers the most injections? Why do you think this line suffers the most injections?
- Use the custom constraints to generate two plots that you find interesting. Explain why you find these plots interesting and provide an analysis of what the plots show.

Based on the above analysis does anything stand out to you? Finally, estimate how, if at all, the added error detector could improve the application’s reliability.