Characterization and Comparison of Application Resilience for Serial and Parallel Executions

Kai Wu[1], Qiang Guan[2], Nathan DeBardeleben[2], Dong Li[1]
University of California, Merced[1]
Ultrascale Systems Research Center, Los Alamos National Lab[2]

USRC · Los Alamos NATIONAL LABORATORY EST. 1943 · UCMERCED

Motivation and Introduction

- **Fault injection can be expensive**
  - Inject a large number of faults to ensure statistical significance.
- **Injecting faults into parallel execution is more expensive**
  - Comparing to serial execution, parallel execution needs more hardware resource.
  - Injecting faults into parallel program is difficult because there is a large exploration space for fault injection.

Research questions:

- Explore the correlation between the serial and parallel executions in term of their resilience.
  - Does the application resilience remain the same across the serial and parallel executions?
  - If not, what code structure causes the difference?

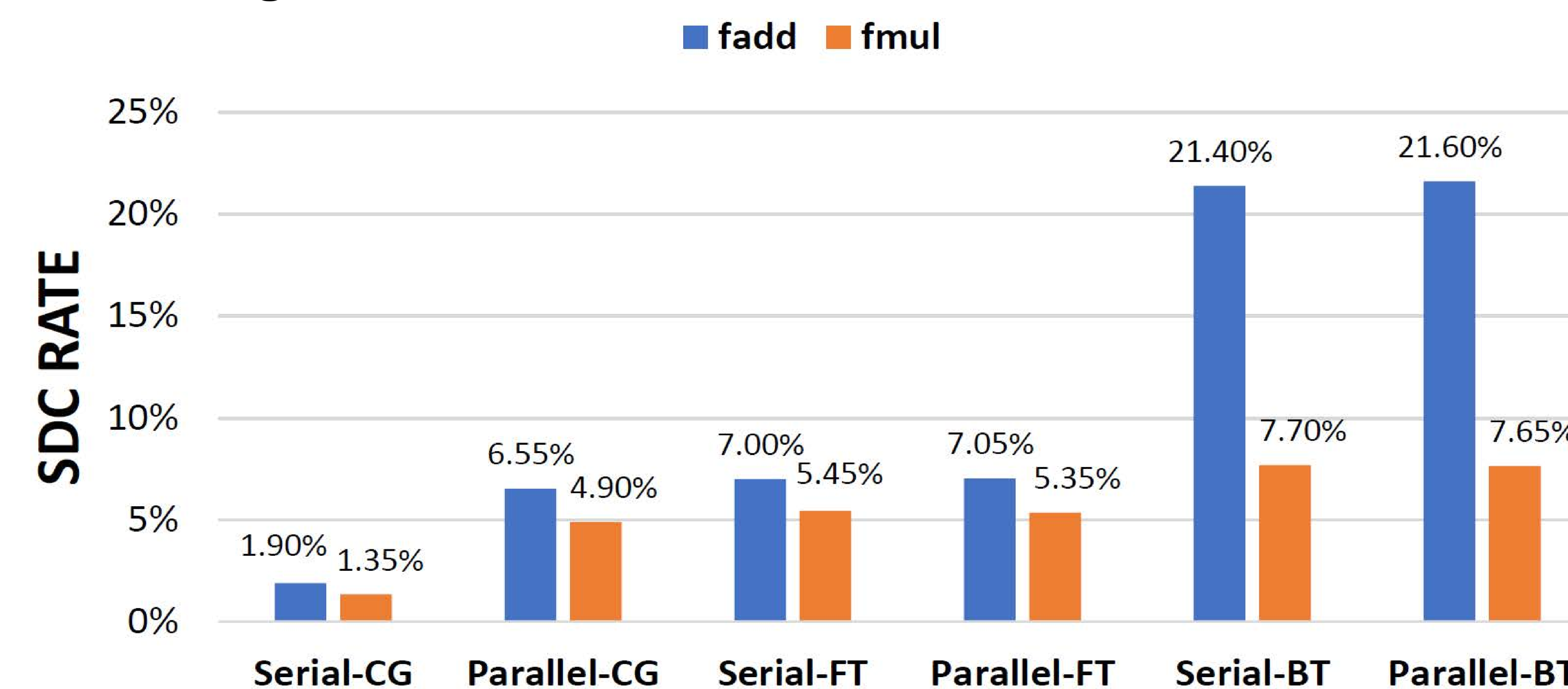
Methodology

- Use a fault injection tool, PFSEFI[1] to study three NAS benchmarks (CG, FT, BT) with the input problem S.
  - Randomly select an instruction and then randomly flip a bit in the operand;
  - Inject faults into the whole applications;
  - Only inject two types fault: floating point (fadd) add and floating point multiplication (fmul);
  - We gradually increase the number of fault injection tests until the fault injection result becomes stable.
- The fault injection results are classified into three types:
  - Benign: the computation results of benchmarks pass the benchmarks' verification phase;
  - Silent data corruption (SDC): benchmarks' verification fails;
  - Crashes: the benchmark cannot run to completion.
- Map the instruction address into the application source code via PYELFTOOLS[2].
- Rank the faulty instructions and analyze the source code.

References:
[1] https://github.com/lanl/PFSEFI
[2] https://github.com/eliben/pyelftools

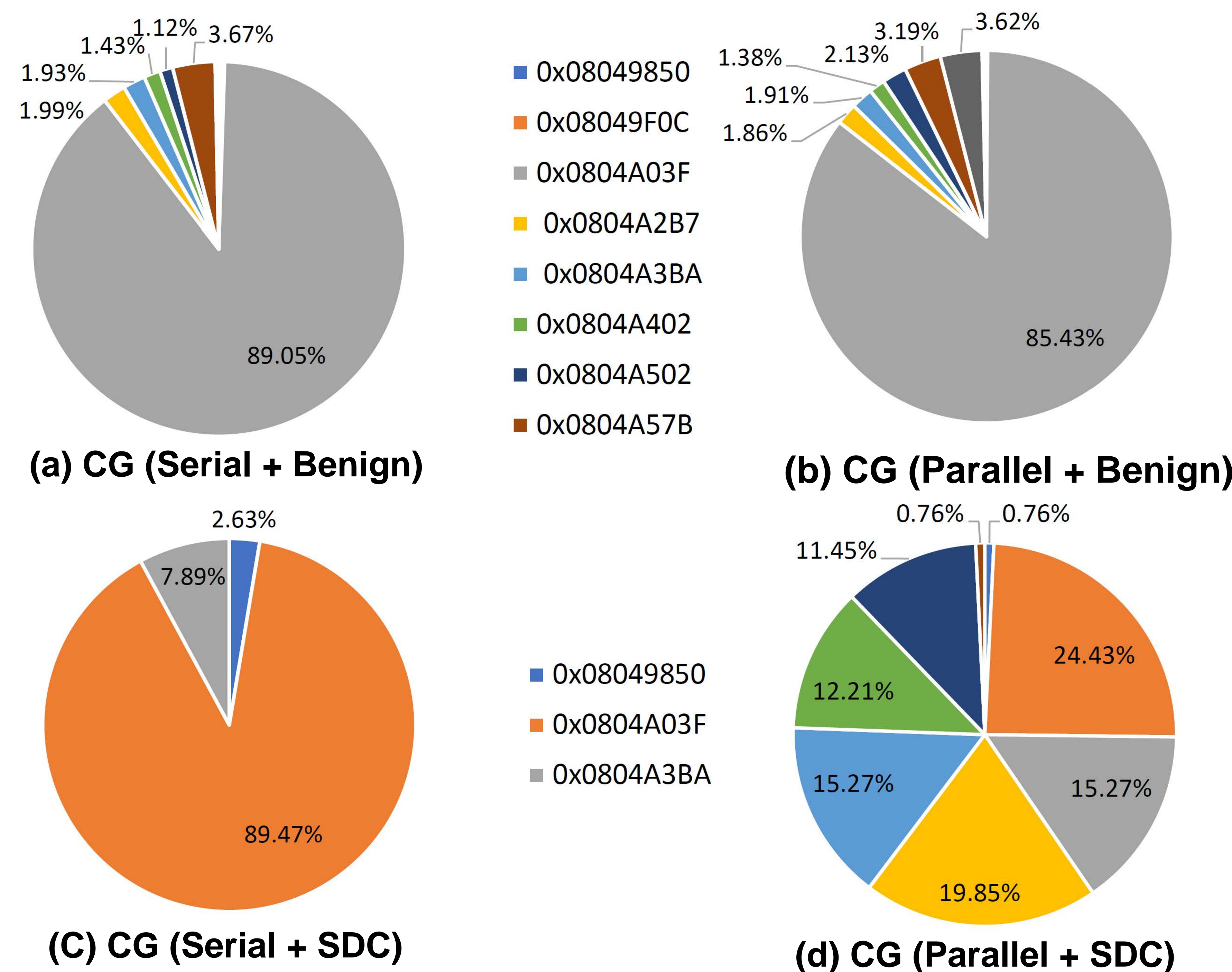
Evaluation Results

Figure 1: SDC rates of NPB CG,FT, BT Benchmarks



We Collect 10,000 fault injection test results for each benchmark and calculate the SDC rate every 1000 tests. The SDC rates become stable after first 6,000 tests and show different similarities for different benchmarks.

Figure 2: The distribution of fadd instructions in CG fault injection tests.



(a) CG (Serial + Benign)

(b) CG (Parallel + Benign)

(C) CG (Serial + SDC)

(d) CG (Parallel + SDC)

Observation 1:
The instruction at 0x0804A03F appears in all cases (serial+benign), (serial+SDC),(parallel+benign) and (parallel+SDC). Such frequent instruction is often be used in the benchmark, and easier to be corrupted.

Figure 3: Source code analysis for the observation 2

Serial code cannot jump into the loop ! This MPI communication does not exist.

```
......
  sum = 0.0d0
  do j=1, lastcol-firstcol+1
    sum = sum + p(j)*q(j)   // EIP 0x0804A2B7 fault injection position
  enddo

// When we use serial code, l2npcols = 0
// When we use parallel code, l2npcols = 1

  do i = 1, l2npcols
    call mpi_irecv( d,
>                 1,
>                 dp_type,
>                 reduce_exch_proc(i),
>                 i,
>                 mpi_comm_world,
>                 request,
>                 ierr )
    call mpi_send(  sum,
>                 1,
>                 dp_type,
>                 reduce_exch_proc(i),
>                 i,
>                 mpi_comm_world,
>                 ierr )
    call mpi_wait( request, status, ierr )
    sum = sum + d
  enddo
......
```

Observation 2:
Some instructions like 0x0804A2B7, 0x0804A3BA, 0X0804A402 and 0x0804A502 only appear in (serial+benign), (parallel+benign) and (parallel+SDC), but do not appear in (serial+SDC).

In Figure 3(0x0804A2B7), the serial and parallel executions have a different value for the variable ***l2npcols***, which leads to different code structure (particularly the MPI synchronization) for serial and parallel executions.

Figure 4: Source code analysis for the observation 3

Serial code cannot jump into the loop !

```
......
......
// When we use serial code, l2npcols = 0
// When we use parallel code, l2npcols = 1

  do i = l2npcols, 1, -1
    call mpi_irecv( q(reduce_recv_starts(i)),
>                 reduce_recv_lengths(i),
>                 dp_type,
>                 reduce_exch_proc(i),
>                 i,
>                 mpi_comm_world,
>                 request,
>                 ierr )
    call mpi_send(  w(reduce_send_starts(i)),
>                 reduce_send_lengths(i),
>                 dp_type,
>                 reduce_exch_proc(i),
>                 i,
>                 mpi_comm_world,
>                 ierr )
    call mpi_wait( request, status, ierr )
    do j=send_start,send_start +
reduce_recv_lengths(i) - 1
      w(j) = w(j) + q(j)          // EIP
0x0804A163 fault injection position
    enddo
  enddo
......
```

Observation 3:
The instruction at 0x0804A163 is only shown in (parallel+bengin) and (parallel+SDC), and such instruction only exists in the parallel execution since the variable ***l2npcols*** has a different value between the serial and parallel executions.

Conclusions and Future Work:

- This work is a preliminary study to explain the reason for similar or different application resilience between the serial and parallel executions.
- For the future work, we will investigate more benchmarks and establish a model to predict application resilience for the parallel execution based on the fault injection results for the serial execution.

LA-UR-17-26370