# LetGo: A Lightweight Continuous Framework for HPC Applications under Failures

**Bo Fang**, Qiang Guan, Nathan Debardeleben, Karthik Pattabiraman and Matei Ripeanu

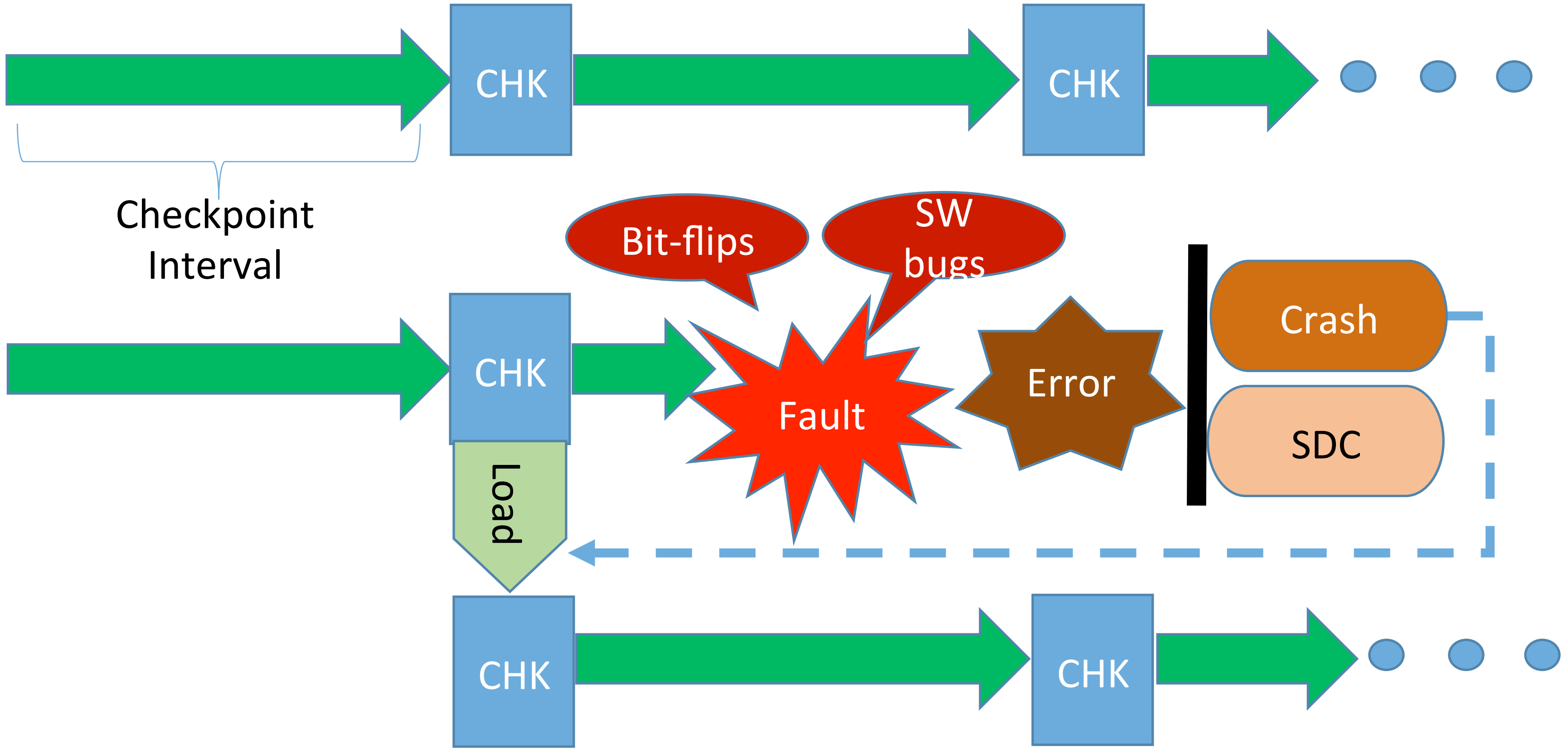
# Resilience is a critical factor for system to scale.


$$MTBF = Total\ Time / Number\ of\ failures$$

$$MTBFsystem = MTBFnode\ /\ Number\ of\ nodes$$

Fault tolerance techniques are needed!

# Classical FT Technique: Checkpoint/Restart




$$Efficiency = \Sigma(Ckpt\ interval) / \Sigma(Ckpt\ interval + Overhead)$$

$$Optimal\ checkpoint\ interval = f(Ckpt\ overhead, MTBF)$$ [**Young**, **Daly**]

# Design Space for Recovery strategies

| | Application-aware | Application-agnostic |
|---|---|---|
| Rollback | [Gamel2013] | [Aupy2013]<br>Classical system-level C/R |
| Forward | [Wu2016] | [Chien2015]<br>LetGo<br>[Bo2017] |

# Why this could work?

Can we detect the error?

OS Exceptions

**After the state is repaired** …

Will the crash happen again?

Corrupted state is contained [Li2015]

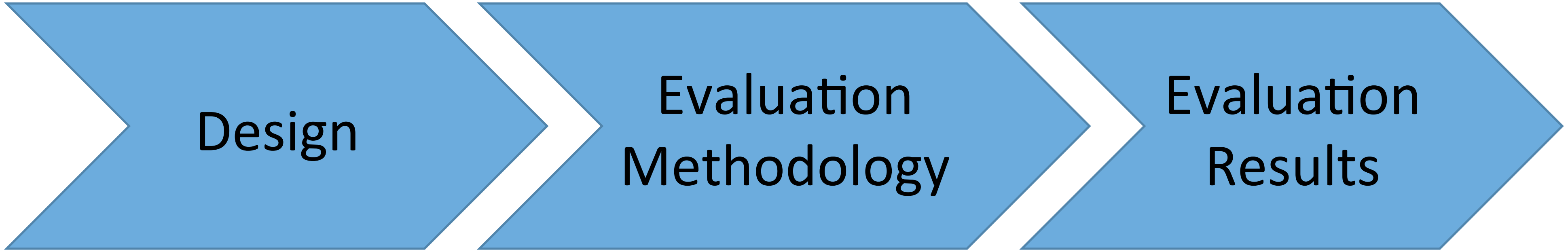
Can we expect correct results?

Applications with convergent behavior

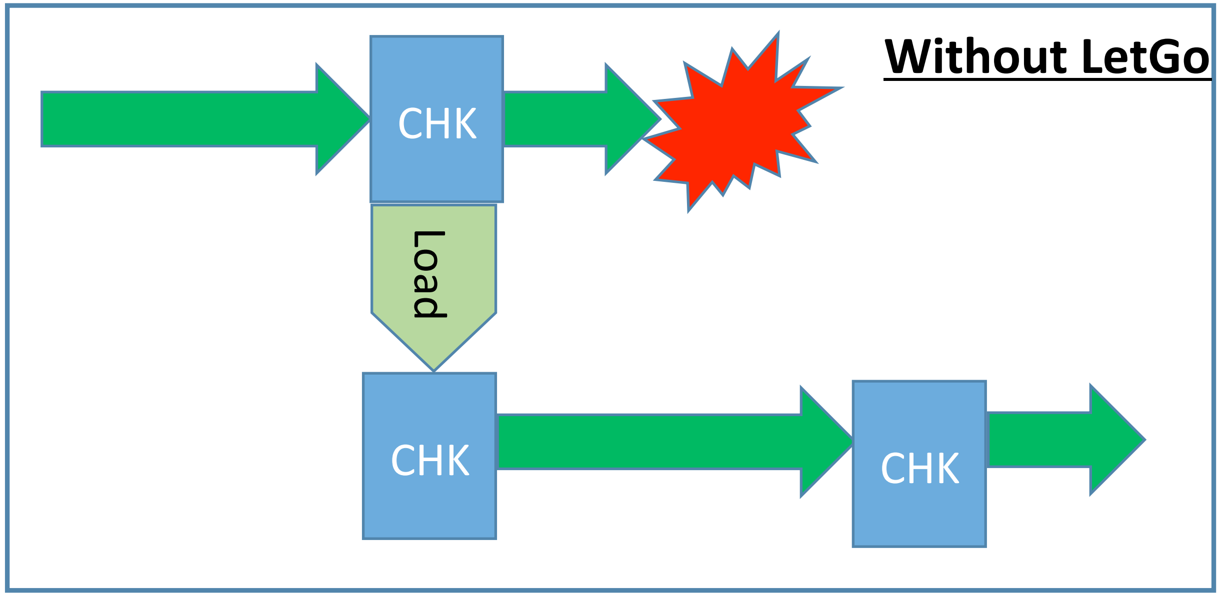
Can we detect incorrect results?

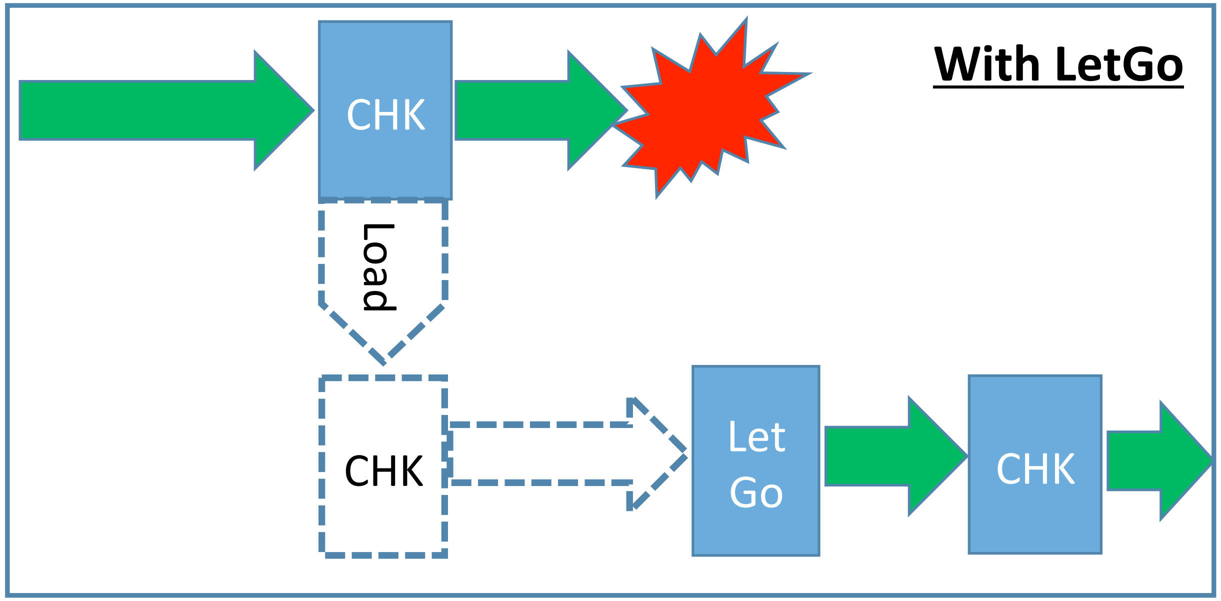
App-specific correctness check

# Our Approach: LetGo



# Forward Recovery with LetGo




Expected Benefits

- **Lower checkpoint frequency**
- Avoid the overhead of restart

Potential risk:

- Higher rate of incorrect results (SDCs)

# LetGo – Design Requirements

## Transparency

- No need to change the application code

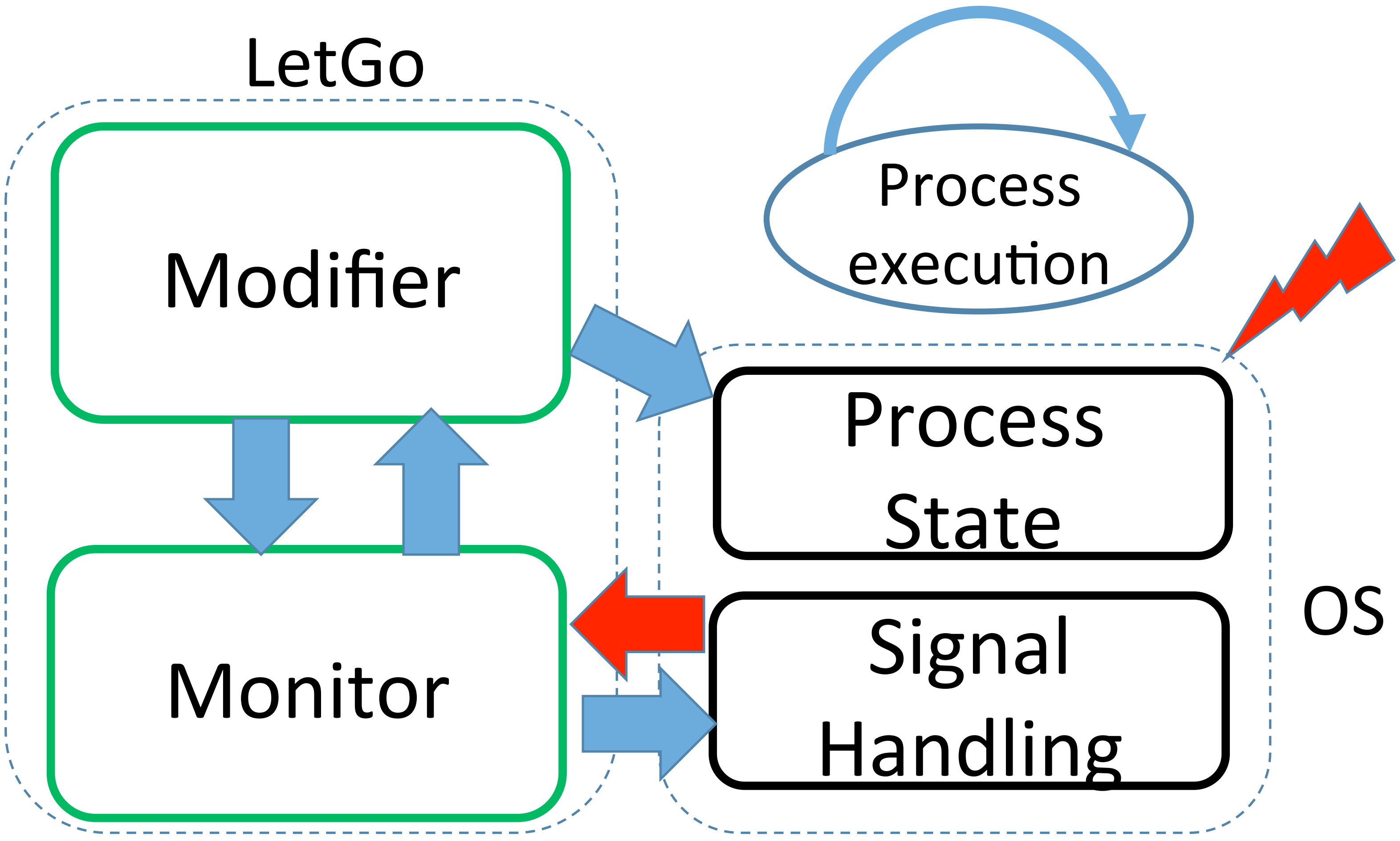
## Convenience

- No need to recompile the application or make changes to the runtime environment

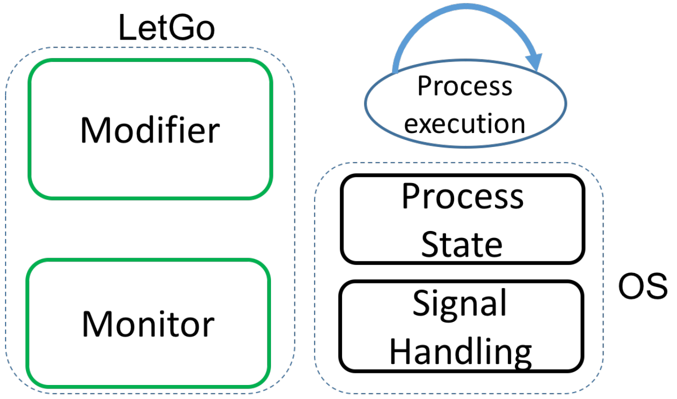
## Low overhead

## Low rate of newly introduced failures

# LetGo - Design Overview



# Monitor




Intercepts OS signals for applications

- Uses GDB to attach to application at launch
- Configures the signal handling (SIGSEGV, SIGBUS, SIGABORT)

Benefits:

No assumption about the compilation level

No change to the application

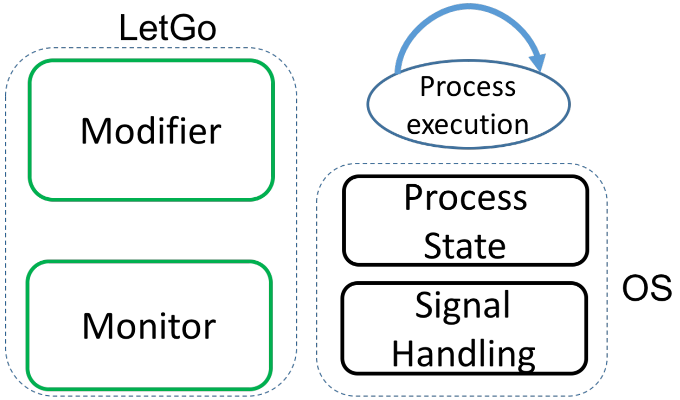
Low overhead ( 0.1%)

# Modifier

Goal: Increase the probability of successful continued execution (low rate of newly introduced failures)

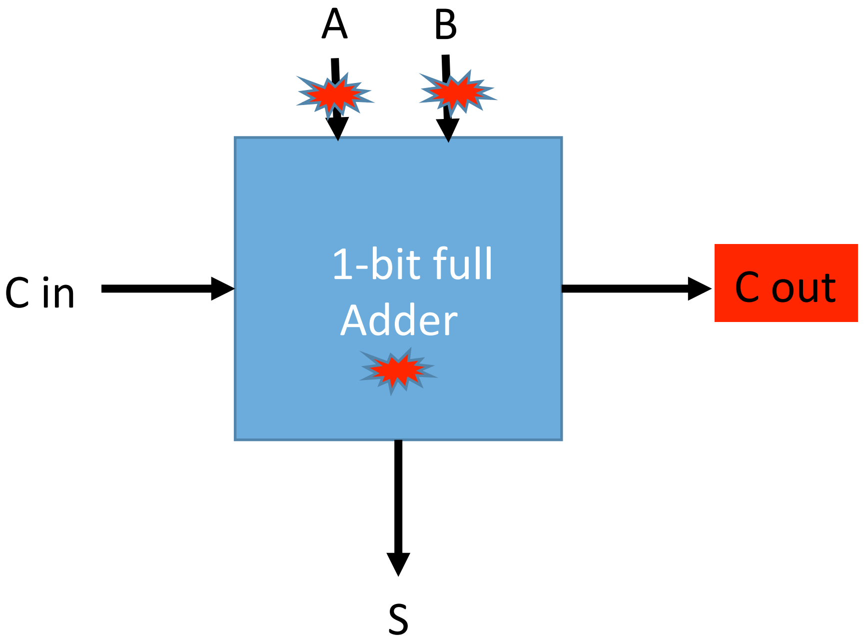
Basic action: move the program counter (PC) after the crash causing instruction

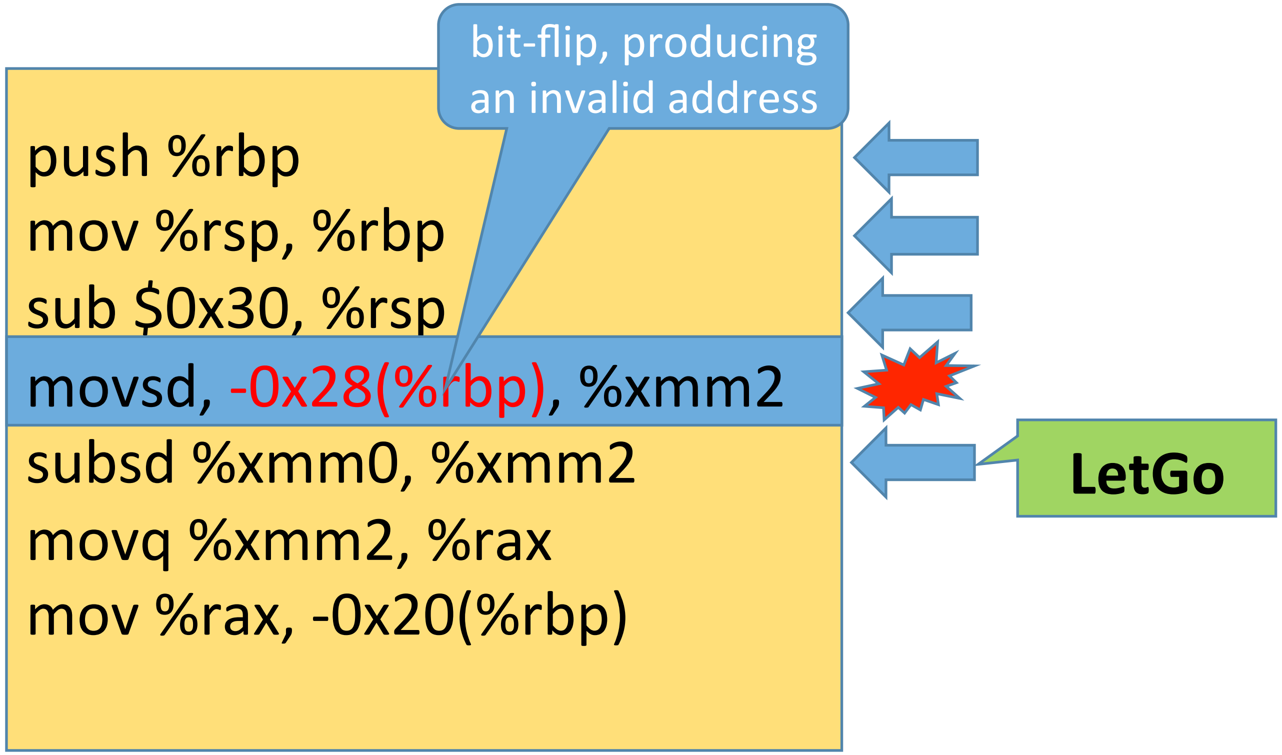
Optional: Heuristics to repair state



# Modifier: example

- A bit flip affects the computation logic



# Evaluation questions

Can we detect the error?

**After the state is repaired**

Will the crash happen again?

Can we expect correct results?

Can we detect incorrect results?

# Evaluation questions

- Q1: What is the chance of the continued execution?
- Q2: What is the chance of successful state repair leading to results?
- Q3: What is the [increase in the] rate of (un)detected incorrect results.
- Q4: What is the performance improvement LetGo brings?

Evaluation methodology

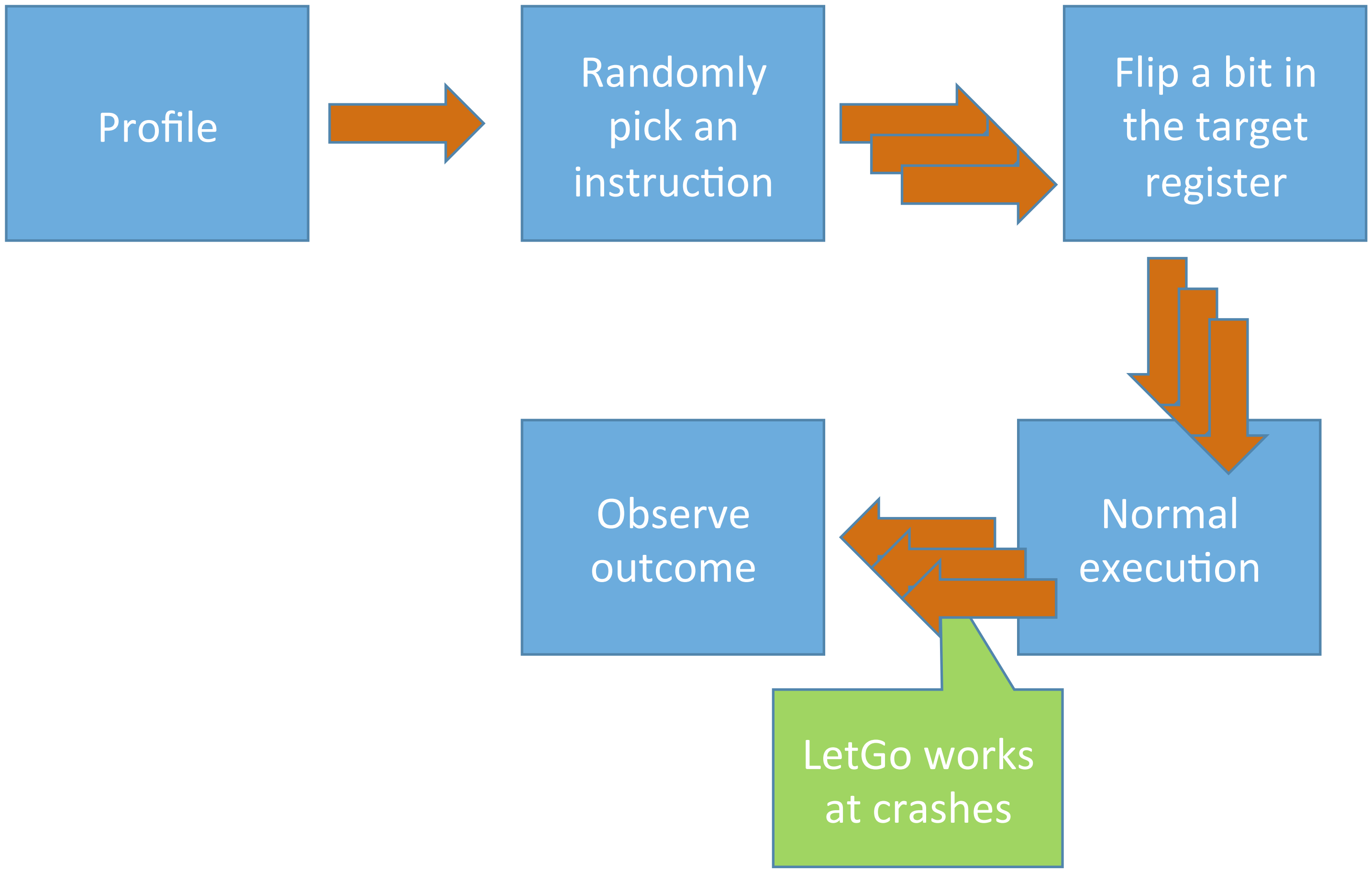
- Fault injection: Q1, Q2, Q3
- Simulations: Q4

# Benchmarks

Convergent behavior

| Application | Domain | # of dyn inst. (billions) | Data to check for SDC | Criteria used in application check |
|---|---|---|---|---|
| LULESH | Hydrodynamics | 1.0 | Mesh | Number of iterations: exactly the same<br>Final origin energy: correct to at least 6 digits<br>Measures of symmetry: smaller than 10−8 |
| CLAMR | Adaptive mesh refinement | 2.8 | Mesh | Threshold for the mass change per iteration |
| COMD | Classical molecular dynamics | 5.1 | Each atom's property | Energy conservation |
| SNAP | Discrete ordinates transport | 1.6 | Flux solution | The flux solution output should be symmetric |
| PENNANT | Unstructured mesh physics | 1.7 | Mesh | Energy conservation |
| HPL | Dense linear solver | 1.2 | Solution vector | Residual check on the solution vector |

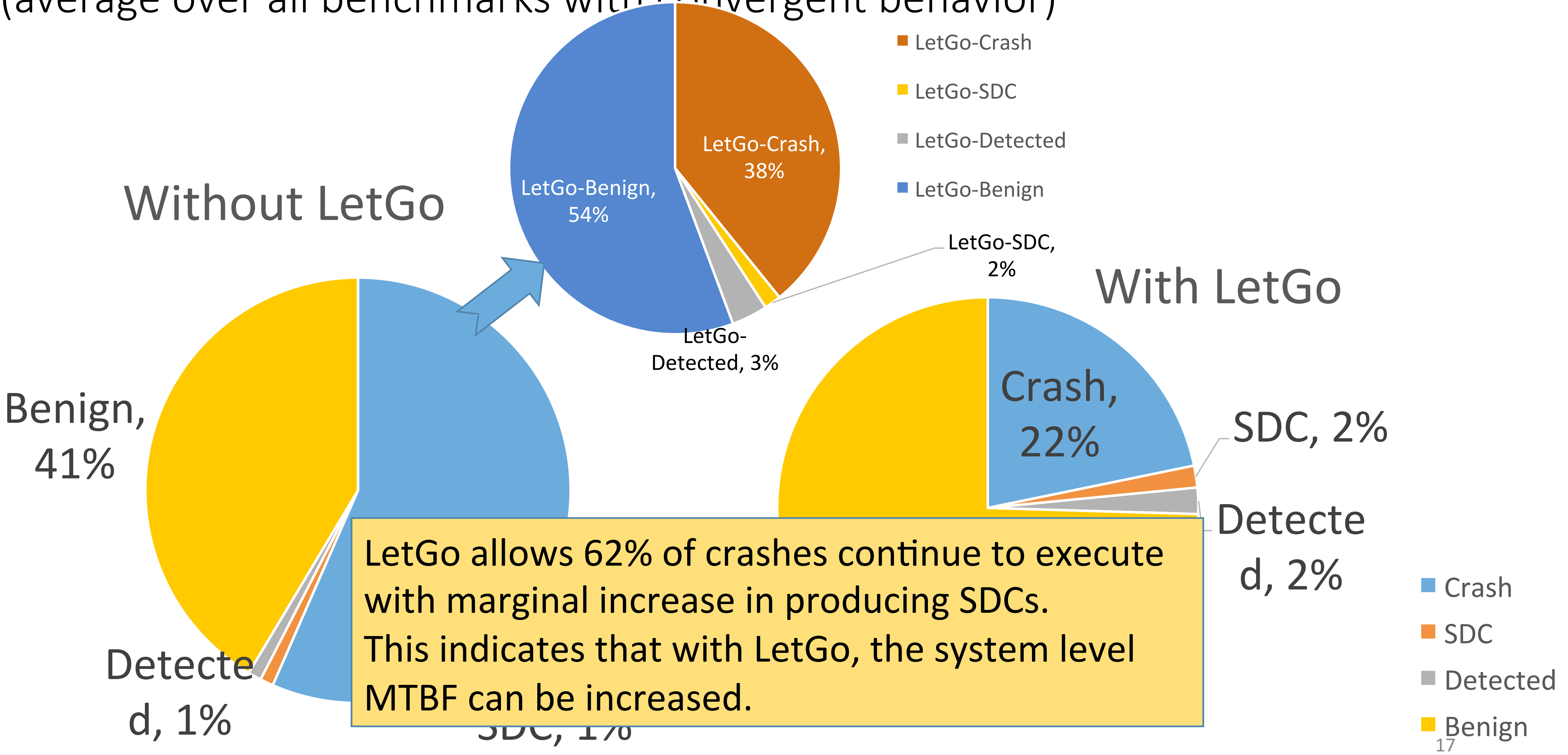
# Fault Injection Methodology




| Outcome | Description |
|---|---|
| Crash | OS exception |
| SDC | Undetected incorrect results |
| Benign | Correct results |
| Detected | Detected incorrect results |

# Evaluation Results

(average over all benchmarks with convergent behavior)



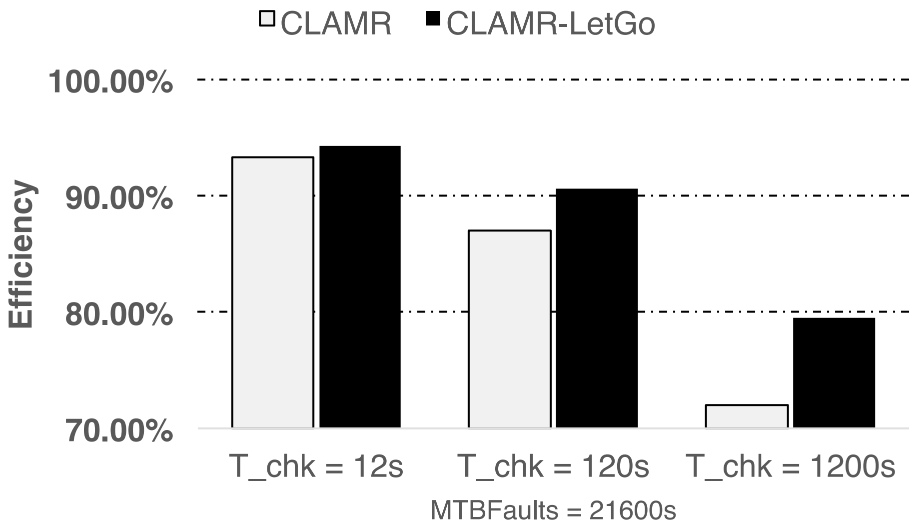

LetGo allows 62% of crashes continue to execute with marginal increase in producing SDCs.
This indicates that with LetGo, the system level MTBF can be increased.

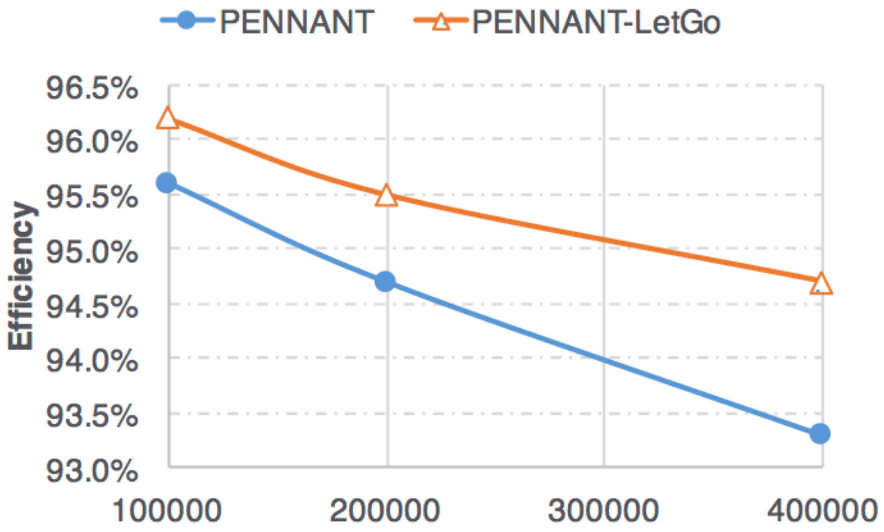
# Evaluation questions

- Q1: What is the chance of the continued execution?
- Q2: What is the chance of successful state repair leading to results?
- Q3: What is the [increase in the] rate of (un)detected incorrect results.
- Q4: What is the performance improvement LetGo brings? What is the impact of system scale?

# Simulation Results

- $Efficiency = Useful\ time / Total\ time$




1. As checkpoint overheads increase, LetGo performance advantage increases
2. As the system scales, the LetGo performance advantage increases

# Conclusion

Can we detect the error?

**After the state is repaired**

Will the crash happen again?

62%
Not crash

Can we expect correct results?

54%
Correct

Can we detect incorrect results?

3%
detected

# Conclusion

- LetGo improves the overall C/R efficiency and leads to a longer checkpoint interval.
- LetGo offers "clean" solution for HPC environment


- Email: bof@ece.ubc.ca
- Github: https://github.com/flyree/LetGo.git

- **Netsyslab UBC:** http://netsyslab.ece.ubc.ca/
- **Dependability lab UBC:** http://blogs.ubc.ca/karthik/

# Modifier: example

bit-flip, producing an invalid address

```
push %rbp
mov %rsp, %rbp
sub $0x30, %rsp
movsd, -0x28(%rbp), %xmm2
subsd %xmm0, %xmm2
movq %xmm2, %rax
mov %rax, -0x20(%rbp)
```

## Challenge I: data corruption

```
…
movsd, -0x28(%rbp), %xmm2
movsd, $0x0, %xmm2
subsd %xmm0, %xmm2
…
```

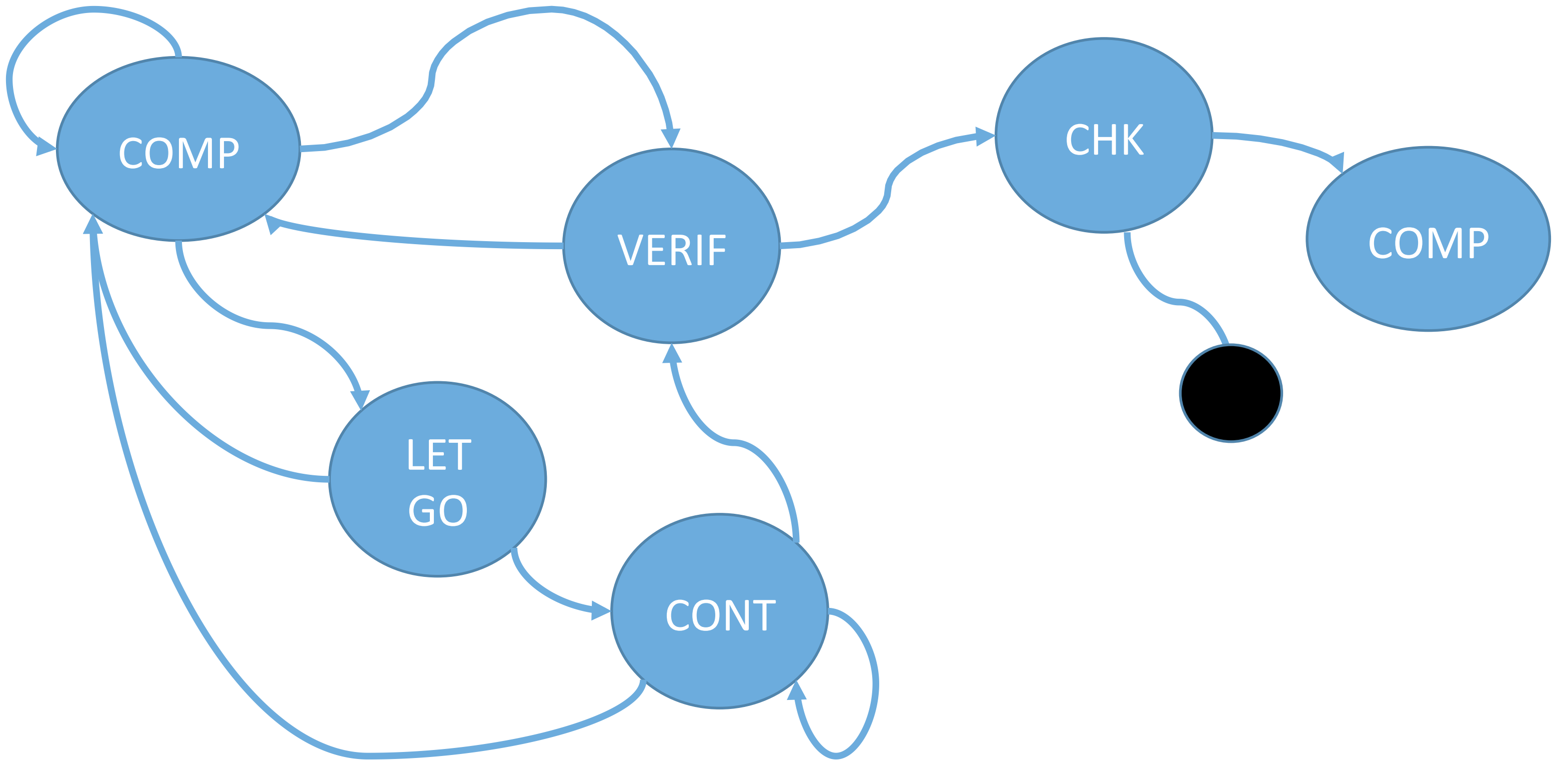
%xmm2 may contain random value

## Challenge II: pointer corruption

```
push %rbp
mov %rsp, %rbp
sub $0x30, %rsp
…
```

%rbp should be within [%rsp, %rsp+30]

# State Machine of a C/R with LetGo



# Computation Convergence Example