

# *diffdbg* - Checkpoint-Based State Comparison for Localizing Correctness Regressions in Scientific Codes

High-Performance  
Computing Center  
Stuttgart

Christoph Niethammer <niethammer@hls.de>



# Motivation

- Optimization work of scientific codes is routine – but may introduce silent bugs or changes in numerical behavior
- The failure mode isn't a crash – it's a plausible but wrong result that passes unnoticed until much later (or never)
- Finding where a divergence first appears – which function, which iteration, which array index – is currently manual: print-statement bisection, ad hoc dumping, or reasoning about the diff by hand
- Example: "Optimizations in an MD code resulted in silent changes – much later resulting in molecules disappearing in some simulations - took days to find cause: a change in force calculation interacted badly with cell based algorithms and halo exchange"

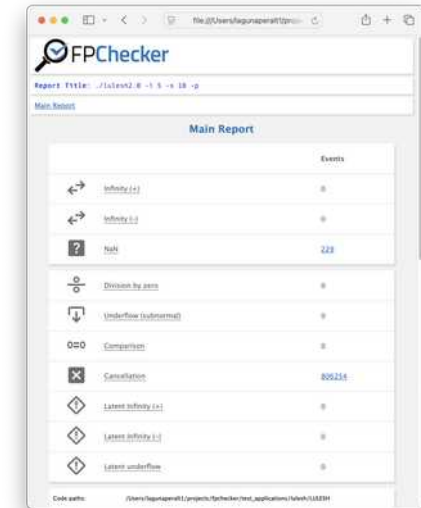


# Core Question

- Given two runs of the same or modified code, do they compute the same thing, and if not, where exactly occur changes?
- Two difficulties:
  - 1) localizing the divergence in code + iteration space
  - 2) distinguishing real bugs from legitimate floating-point non-associativity

# Existing solutions – numerical debuggers

- CADNA: A tool used to detect numerical Instability points in simulations, caused by floating point arithmetic and inaccuracies
- Verificarlo: Tool to emulate different floating point formats in program execution
- FPChecker: profiling of floating point behaviour in HPC applications reporting e.g., rounding hotspots



The screenshot shows the FPChecker web interface. At the top, it says 'FPChecker' with a magnifying glass icon. Below that, it says 'Report Title: ./latency2\_E -i 5 -s -e 18 -p'. There is a 'Main Report' section with a table of events. The table has two columns: 'Events' and 'Count'. The events listed are: 'Infinity (x)', 'Infinity (y)', 'NaN', 'Division by zero', 'Underflow (subnormal)', 'Comparison', 'Cancellation', 'Latent infinity (x)', 'Latent infinity (y)', and 'Latent underflow'. The counts for these events are: 0, 0, 228, 0, 0, 0, 809214, 0, 0, and 0 respectively.

| Events                | Count  |
|-----------------------|--------|
| Infinity (x)          | 0      |
| Infinity (y)          | 0      |
| NaN                   | 228    |
| Division by zero      | 0      |
| Underflow (subnormal) | 0      |
| Comparison            | 0      |
| Cancellation          | 809214 |
| Latent infinity (x)   | 0      |
| Latent infinity (y)   | 0      |
| Latent underflow      | 0      |

source: [fpchecker.org/](https://fpchecker.org/)

## What they do :)

do quantify rounding-error propagation within a single run

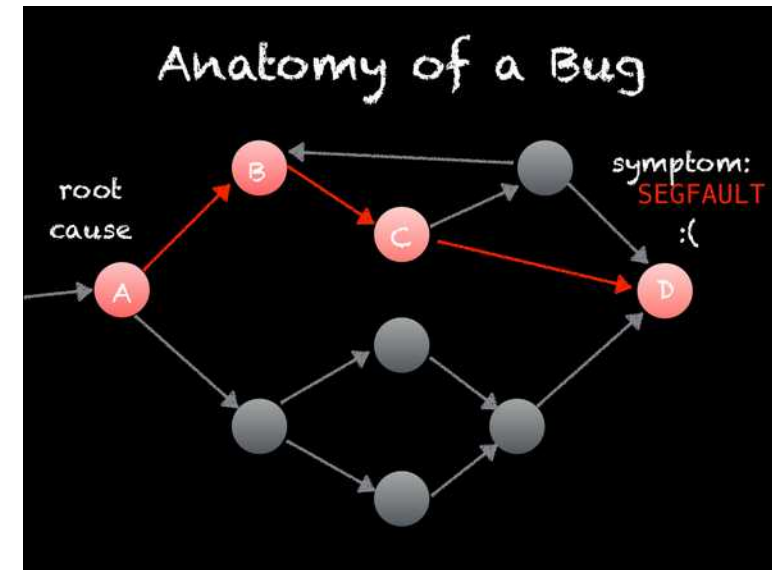
## What they don't :(

don't compare two executions against each other

## Existing solutions – record & replay

- rr (Record and Replay)

excellent for finding the exact divergent instruction once you know roughly where to look



Source: [fitzgen.com/2015/11/02/back-to-the-future.html](http://fitzgen.com/2015/11/02/back-to-the-future.html)

### What it does :)

deterministic replay + reverse execution

### What it doesn't :(

- doesn't compare two executions against each other
- no notion of domain semantics (only memory watchpoints)

## Existing solutions – interactive differential debuggers



- IDD (Interactive Differential Debugging)

## What it does :)

Runs baseline + regressed executable  
side by side under synchronized  
gdb/lldb sessions

## What it doesn't :(

- provide performance - dual-session, continuous debugger-control overhead, and developer manually drives both sessions and issues comparison commands
- no support for parallel applications

```
controller-
1 (int) a_var1 = 04
2 (int) a_var2 = 04
ccktrace
1 frame #0: 0x0000555555554914 .out'a arg1 1 frame #0: 0x0000555555554914 .out'a[5/
2=1, arg2=2) at b.cxx:7
2 frame #1: 0x0000555555554964 .out'main a 2 frame #1: 0x0000555555554964 .out'main
:14 t b.cxx:14
3 frame #2: 0x00007ffff767cb97 libc.so.6 3 frame #2: 0x00007ffff767cb97 libc.so.6
art_main(main=.out'main at b.cxx
libc_start_main(main=.out'main at b.cxx
argc=7, argv=0x00007ffff767cb97, in
available), fini=cunavailable), r
rtl
als
2 frame #0: 0x0000555555554914 .out'a( 45 frame #0: 0x0000555555554914 .o[6/
a arg1=1, a
g2=2) at b.cxx:7
4 4 #include <iostream> 47 4 #include <iostream>
5 5 48 5
6 6 int a(int arg1, int arg2) { 49 6 int a(int arg1, int arg2) {
7 -> 7 int a_var1 = 1; 50 -> 7 int a_var1 = 2;
8 8 int a_var2 = 2; 51 8 int a_var2 = 1;
f
d #1, name = 'a.out', stop reason = br[3/43] * thread #1, name = 'b.out', stop reason = b[3/
me #0: 0x0000555555554914 a.out'a arg1=1, as 1 thread #1, name = 'b.out', stop reason = b[3/
t a.cxx:7 frame #0: 0x0000555555554914 b.out'a arg1=1,
#include <iostream> rg2=2) at b.cxx:7
4 #include <iostream>
5
6 int a(int arg1, int arg2) { 6 int a(int arg1, int arg2) {
7 int a_var1 = 1; -> 7 int a_var2 = 2;
8 int a_var2 = 2; 8 int a_var1 = 1;
b 1 6 lldo 2
asi0:diffwindow 'controller' 13:00 03-Jan
```

source: IDD--a platform enabling differential debugging

## Existing solutions – offline diffing

- using diff tools, e.g., h5diff, numdiff, ... on manually dumped program state



### What they do :)

compare dumped output files with  
numeric tolerance support

### What they don't :(

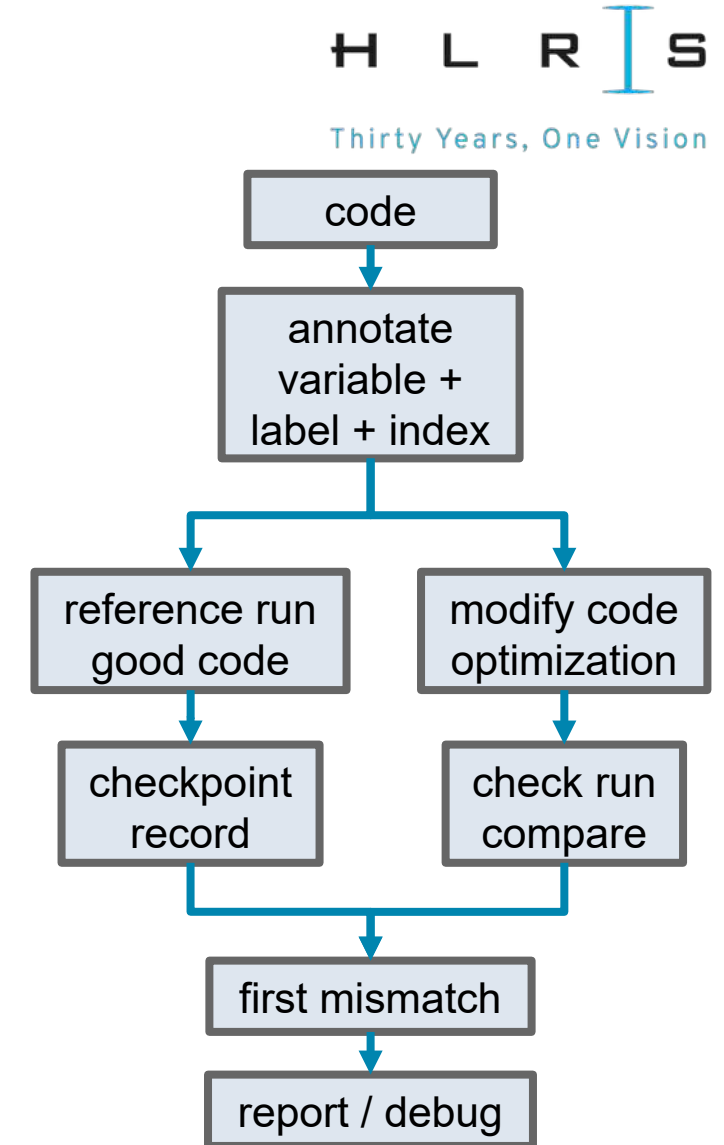
- They are post-run – no debugger integration
- typically only practical at coarse granularity

# Goals for diffdbg

- Comparison of program executions based on domain-meaningful checkpoints
- Simple to use
- Low integration cost (header-only, simple drop into existing code)
- Cheap enough to leave (compiled) in permanently
- Different strategies for data validation  
e.g., full or statistical data validation; distinguishing real regressions from floating-point noise (tolerance)
- Should work in nested/recursive call chains

# Core mechanism

- Register/annotate a variable at a checkpoint  
a checkpoint is identified by a user-provided label (string) + multi-dimensional integer index (step, substep, rank, ...)
- Two step analysis process:
  - 1) Reference run recording values from “good application”
  - 2) Check run with modified application for validation
- Step 2 detects mismatches based on different, user-selectable comparison strategies
- On mismatch: report label/index/value
- Optionally trap into an attached debugger at that exact point



# diffdbg basic code annotation

```
#include "diffdbg.hpp"
#include <vector>

int main() {
    diffdbg::init(
        diffdbg::modeFromEnv(),
        diffdbg::fileFromEnv(),
        diffdbg::strategyFromEnv()
    );

    const size_t N = 1000;
    std::vector<double> velocity(N), pressure(N);

    for (int step = 0; step < 5; ++step) {
        ...
        DIFFDBG_CHECK_VEC(velocity, step);
        DIFFDBG_CHECK_VEC(pressure, step);
    }

    diffdbg::finalize();
    return 0;
}
```



Header file



Initialization (configuration parameters)



Annotation, perform checkpoint record / check  
DIFFDBG\_CHECK\_VEC(string label, long idx1, ...)



Cleanup (flushing checkpoint records)

- Step 1: Record checkpoints

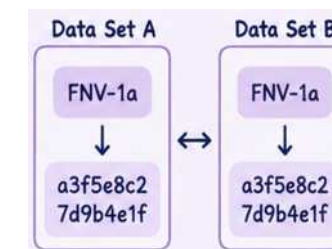
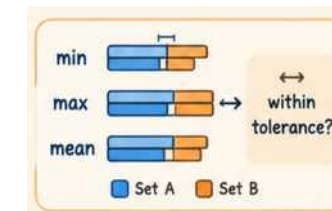
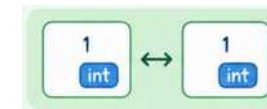
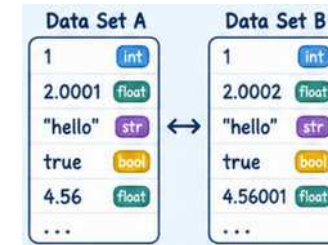
```
DIFFDBG_MODE=record DIFFDBG_FILE=ref.bin ./example  
[diffdbg] mode=record file=ref.bin strategy=full  
[diffdbg] wrote 10 reference entries to ref.bin (strategy=full)
```

- Step 2: Run comparison

```
DIFFDBG_MODE=check DIFFDBG_FILE=ref.bin ./example  
[diffdbg] mode=check file=ref.bin  
[diffdbg] MISMATCH label='velocity' idx=(3) index=42 ref=0.0994281 cur=0.100428  
[diffdbg] trapping into debugger  
          (attach with a debugger, or run the whole process under gdb/rr,  
            to catch this)
```

# Comparison strategies

- **full:**  
full data-set comparison using configurable tolerance for floating point values and types' equality operator for non-float types.
- **first:**  
check first element in checkpoint based on same method as for full
- **stats:**  
low memory overhead comparison using min max and mean values in combination with floating point tolerances
- **hash:**  
low-memory-overhead hash-based comparison based on FNV-1a hash function



## Example: Index-off

e.g., when changing from AoS to SoA:

```
std::vector<double> coords(3 * N);  
coords[i+2] += dt * velocity_z[i];
```



```
std::vector<double> coords(3 * N);  
coords[i] += dt * velocity_z[i];
```

$2*N+i$

**Symptom:** velocity update silently lands in the wrong component – positions drift in a plausible-looking but physically wrong way.

**diffdbg** catches it in (full, first, hash, stats) strategies as values diverge from the reference run

## Example: Loop bound of-by-one

e.g., when performing cache blocking:

before

```
for (int i = 0; i <= numElem; ++i) {  
    p[i] = EvalEOS(e[i], v[i]);  
}
```



after

```
for (int t = 0; t <= numElem; t += TILE) {  
    int end = min(t + TILE, numElem);  
    for (int i = t; i < end; ++i) {  
        p[i] = EvalEOS(e[i], v[i]);  
    }  
}
```

<=

**Symptoms:** p's last element gets skipped in update

**diffdbg** catches it with (full, hash) strategies as last value diverges from the reference run and likely in (stats) strategy if the missed element differs from the average value

## Example: Rounding issues (casting)

```
double ComputeTimeStep(double soundSpeed, double charLength) {  
    return charLength / soundSpeed;  
}
```



```
float ComputeTimeStep(double soundSpeed, double charLength) {  
    return charLength / soundSpeed;  
}
```

**Symptoms:** dt accululates slowly ...

**diffdbg** can catch it with (hash) strategy as scalar value diverges (in binary representation), with (full, first, stats) no exact position (depending on tolerance settings)

# Scoped context propagation

- diffdbg provides **context scopes** for applications using nested functions and loops, e.g., to take checkpoints several calls deep in a solver
- Implementation uses a RAII-based context stack
- Checkpoints use context stack based index if no explicit index is provided

# diffdbg scope annotation

```
void solveInnerKernel(std::vector<double>& velocity) {
    std::vector<double> residual(N);
    ...
    // tagged with what's on the scope stack
    DIFFDBG_CHECK_VEC(residual);
}

int main() {
    diffdbg::init(...);
    const size_t N = 100;
    std::vector<double> velocity(N);

    for (int step = 0; step < 3; ++step) {
        DIFFDBG_SCOPE(step); // push `step`
        ...
        // flat-loop style: explicit single index
        DIFFDBG_CHECK_VEC(velocity, step);

        for (int substep = 0; substep < 2; ++substep) {
            DIFFDBG_SCOPE(substep); // push `substep`
            ...
            solveInnerKernel(velocity);
        }
    }

    diffdbg::finalize();
    return 0;
}
```



Checkpoint using scope stack as index (step,substep)



Add step to scope stack (step)



Checkpoint with explicit index (step)

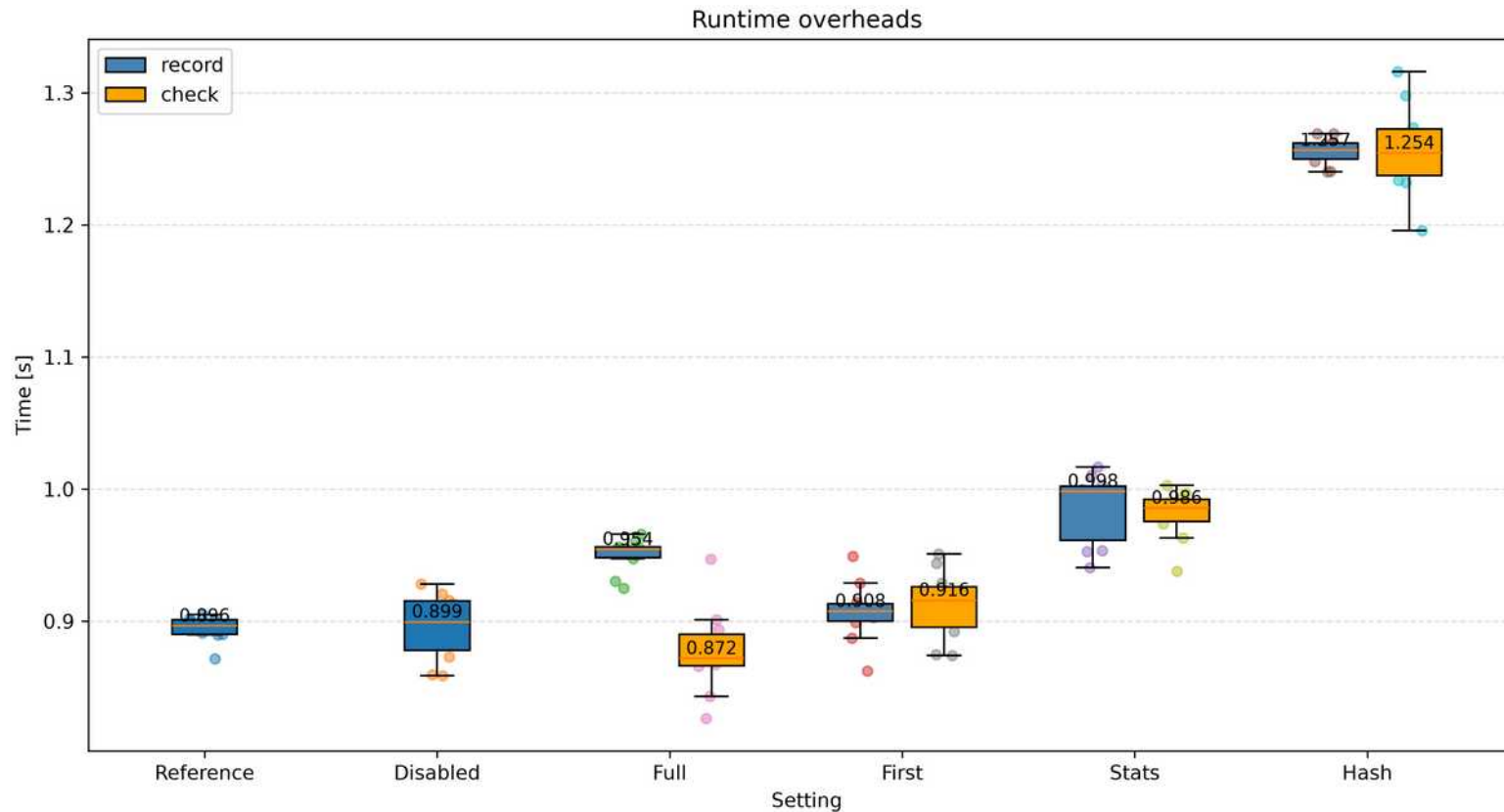


Add substep to scope (step, substep)

# Comparison strategies' overheads

| strategy | Runetim Overhead | Memory Overhead |
|----------|------------------|-----------------|
| full     | $O(N)$           | $O(N)$          |
| first    | $O(1)$           | $O(1)$          |
| stats    | $O(N)$           | $O(1)$          |
| hash     | $O(N)$           | $O(1)$          |

# Runtime Overheads for LULESH



| strategy | record [%] | Check [%] * |
|----------|------------|-------------|
| full     | 5,41       | -3,68       |
| first    | 0,22       | 0,77        |
| stats    | 10,05      | 8,74        |
| hash     | 38,89      | 38,82       |

\*) check mode was run  
without introduced errors

# Pitfalls using diffdbg

- **Adaptive time stepping and refinement algorithms:**

adaptive time step algorithms change whenever the two executions trigger different adaptations, e.g., time step length, making checkpoints non-comparable

- **Using checkpoints inside OpenMP parallel loops:**

diffdbg uses at the moment internal thread locking to protect state introducing high overheads if used inside OpenMP parallel loops/regions



## Further work

- Struct/complex-type-aware tolerance
- CI workflow (GithubAction, ...) for simple integration into CI/CD
- Improved integration with debuggers/RR
- Add memory consumption statistics
- Improve (full) checkpoint efficiency – intermediate flush with limited buffer space
- Other languages (Fortran, ...)?

# Conclusion

- Diffdbg – easy to use tool for differential debugging of C++ codes
- Different strategies available to identify different programming errors: (full, first, stats, hash)
- Small overhead with (first) strategy
- LULESH shows easy adaptation to real applications and manageable overheads

# *diffdbg* - Checkpoint-Based State Comparison for Localizing Correctness Regressions in Scientific Codes

High-Performance  
Computing Center  
Stuttgart

Christoph Niethammer <niethammer@hls.de>

