# A Practical Fine-grained Profiler

Shasha Wen, **Xu Liu**
College of William and Mary
{swen, xl10}@cs.wm.edu

Milind Chabbi
HPE Lab
milind.m.chabbi@hpe.com

# Motivation: Computation Redundancy

```
a = m;
b = sqrt(a);
c = m;
d = sqrt(c);
```

redundant function calls

```
x = a * b;
c = a;
d = b;
y = c * d;
```

redundant operations

# Limitation of Compilers

- Redundancy due to aliasing

```
1.  int  AliasRedundancy(int * a, int *b, int * c, int * d){
2.          int v1 = *a + *b;
3.          int v2 = *c + *d;
4.          return v1 + v2;
5.  }
```

a and c alias each other;
b and d alias each other;

- Redundancy sensitive to path

```
1.  void  PathSensitiveRedundancy(){
2.  BB1:      v1 = a + b;
3.            if (cond){
4.  BB2:          c = a;
5.                d = b;
6.            }
7.  BB3:      v2 = c + d;
```

BB1 -> BB3 no redundant!

BB1 -> BB2 -> BB3 redundant

- Redundancy sensitive to inputs

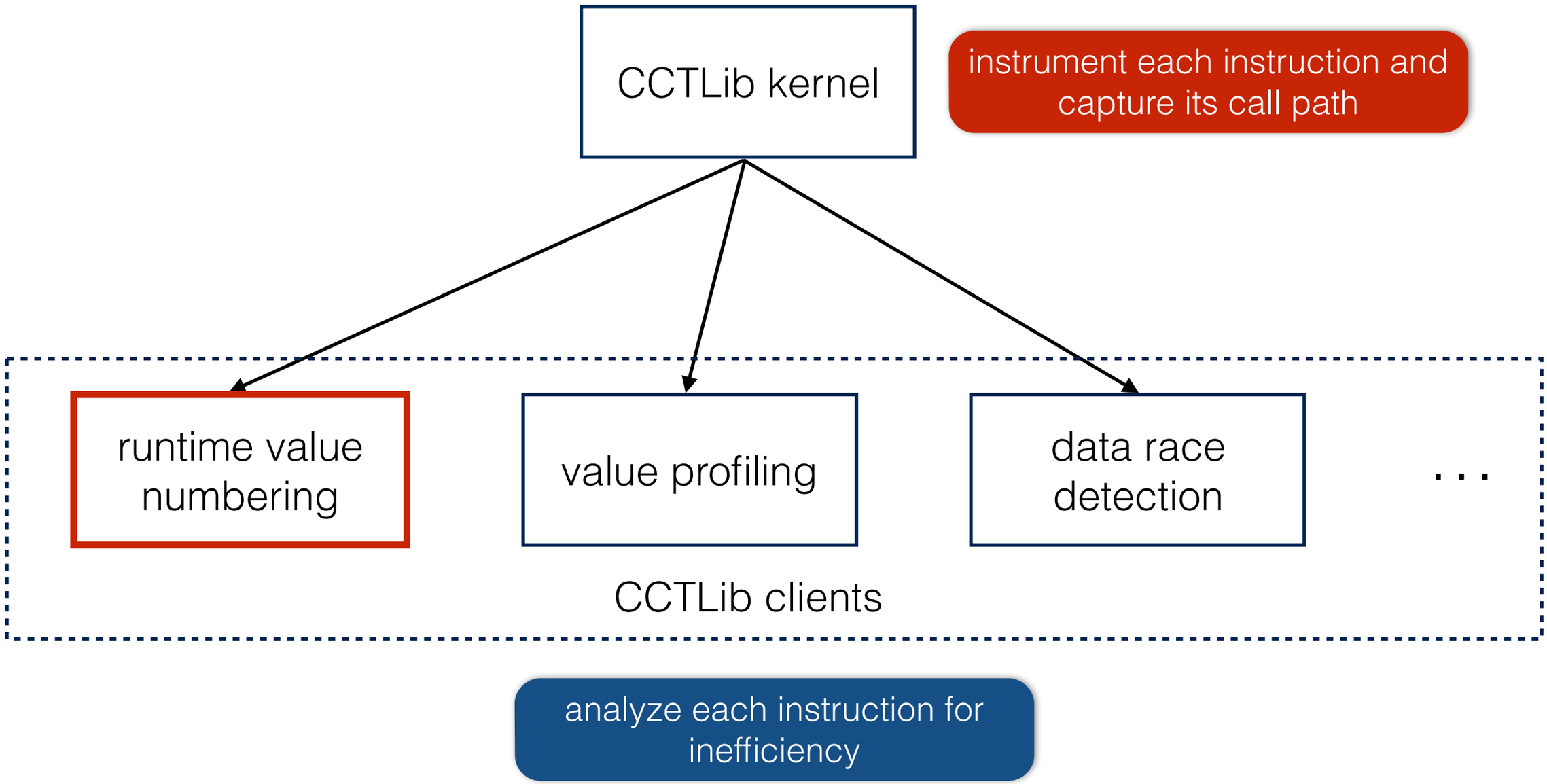
# Our Solution: Fine-grained Profiling

monitor each instruction and operand → identify redundancy

collect call path information → locate redundancy

**CCTLib: A Pin-based Fine-grained Profiler**

# CCTLib Overview



# Runtime Value Numbering

add R1, [mem1] → R1 = R1 + [mem1]

1. Get value numbers of rhs operands
2. Get unique id of operator
3. Hash computation, check redundancy

hash <+, 2, 3> → **key** ∈ ?

history of earlier computation:
map<hashKey, valueNumber>

- Yes! Redundant! -> get calling context -> record redundant pair
- No, insert <**key**, newValueNumber> to map

5. Update value number of target

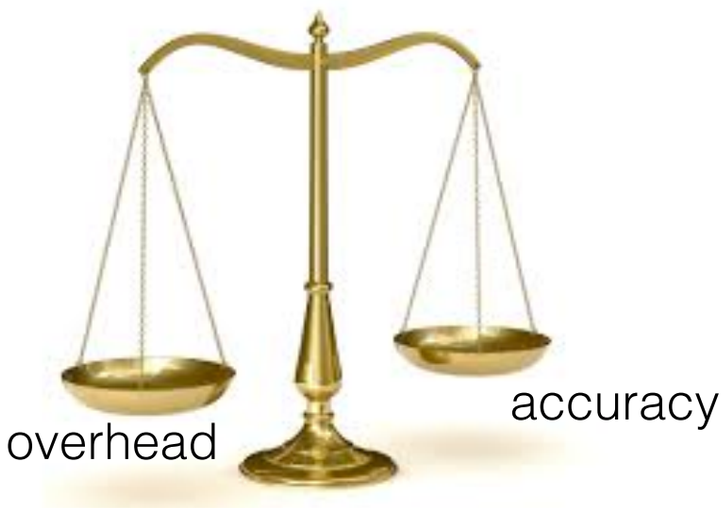
- If redundant: VN(R1) = VN(old computation)
- else : VN(R1) = newValueNumber

# RVN-Methodology

Time overhead can be more than 10000x!
Space overhead can consume all memory!

# RVN-Reducing Time Overhead

- Why?
  - instrument every instruction, check all instances
  - collect the calling context
- How?
  - selective instruction instrumentation
  - bursty sampling
    - enable interval(E)
    - disable interval(D)
  - approximate hashing with collision 0.095%



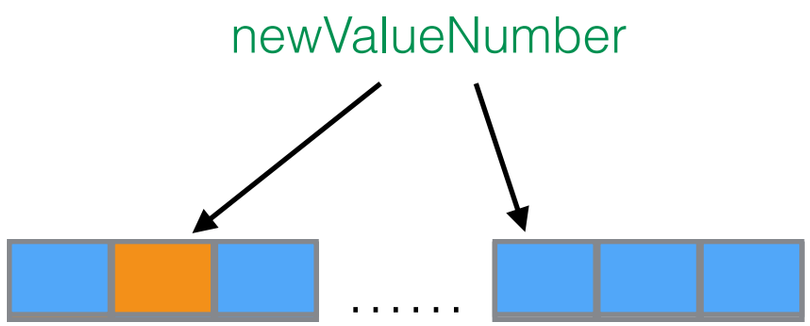
# RVN-Reducing Space Overhead

3 Hash computation, check redundancy

hash < → **key** ∈ history of earlier computation: map<hashKey, valueNumber>

To reduce size: limit "**N**"

add R1, [mem1] (static instruction) → 1 … **N** dynamic instances




store recent N value numbers

- If empty: insert to the slot
- If full : replace the oldest one

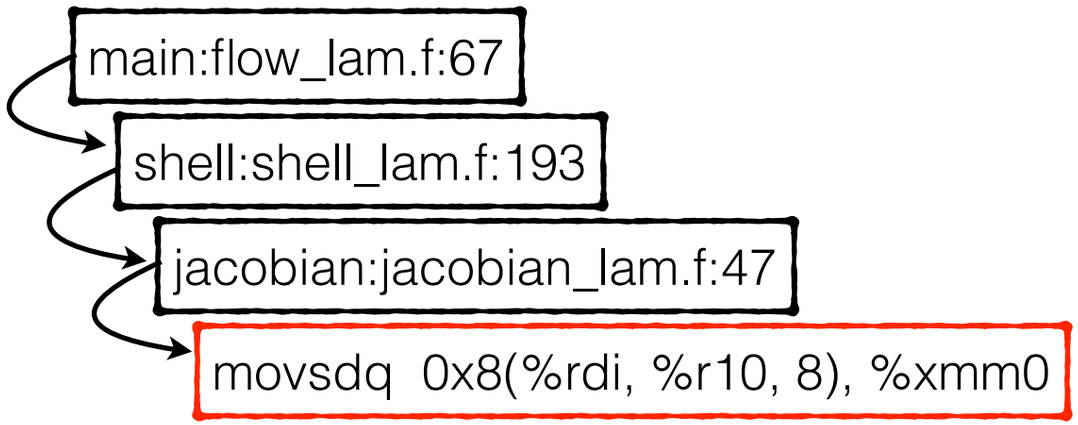
**N** is a parameter;
large N is unnecessary

# RVN-Quantify Redundancy

- Redundancy metrics

$$\hat{R} = \frac{\sum_{i=0}^{N} Redundant\ Instructions\ in\ Sample\ i}{\sum_{i=0}^{N} Total\ Instructions\ in\ Sample\ i}$$

- Calling context
  - redundant function calls
  - path sensitive redundancies




- Redundant pair
  - merge redundancies to the same source line
  - show top redundant pair with high occurrence

RVN provides guidance; developers are responsible for optimization

# Redundant Fraction & Overhead

- Bursty sampling
  - we sampled a window of 1M instructions in every 1B instruction window

| Program | %Redundant Fraction | | | Overhead |
|---|---|---|---|---|
| | Min | Max | GeoMean | |
| gcc | 18.4 | 21.28 | 20.23 | 35.45x |
| hmmer | 18.42 | 20.27 | 19.51 | 36.40x |
| bwaves | 6.61 | 15.09 | 9.97 | 35.6x |
| zeusmp | 4.29 | 4.84 | 4.55 | 27.6x |
| povray | 20.18 | 20.19 | 20.18 | 24.72x |
| applu | 4.67 | 16.11 | 7.9 | 48.78x |

| Program | #procs | %R | Overhead |
|---|---|---|---|
| Sweep3d | 8 | 1.3 | 39.4x |
| LULESH | 27 | 12.18 | 59.23x |
| LAMMPS | 8 | 23.8 | 18.73x |
| Sphot | 8 | 12.37 | 37.11x |
| NPB-BT | 9 | 5.93 | 48.69x |
| NPB—IS | 8 | 25.6 | 10.32x |

- Average redundancy for all the SPEC benchmarks is ~10%
- Number of processors have little influence on the redundancy
- Time overhead = call path collection(~30x) + Runtime Value Numbering(~10x)

# Case Studies

- RVN work on fully optimized code

| Program | Procedure:loops | gcc(-02) | | | | gcc(PGO) | | | icc(-02) | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | %Cycle | %Ins | WS | PS | %Cycle | %Ins | WS | %Cycle | %Ins | WS |
| bwaves | block_solver.f:loop(167) | -97.1 | -97.3 | 1.07x | 0.11 | -96.9 | -97.1 | **1.12x** | -55.1 | -52.6 | 1.04x |
| | jacobian_lam.f:loop(30) | -16 | -16.8 | | | -16 | -16.8 | | -6.3 | -2.9 | |
| | shell_lam.f | -13.6 | -15.2 | | | -10.9 | -13.5 | | -9.1 | -3.9 | |
| hmmer | hmmcalibrate.c:loop(499) | -13.9 | -7.2 | 1.06x | 0.2 | -6.5 | -15.8 | 1.06x | -5.9 | -8.8 | **1.09x** |
| | fast_algorithms.c:loop(119) | -14.1 | -16.9 | | | -7.3 | -15.7 | | -6.1 | -8.9 | |
| Sweep3d | sweep.f:loop(397) | -26.3 | +4.3 | 1.08x | 0.01 | -21 | -9.2 | 1.05x | -39.6 | -6.2 | **1.22x** |
| NPB-MG | mg.f:loop(609) | -12.1 | -2.9 | 1.09x | 0.05 | -21.1 | -12.3 | **1.17x** | — | — | — |
| | mg.f:loop(539) | -14 | -9.1 | | | -19.2 | -5 | | — | — | |

WS: Whole Speedup
PS: Power Saving

# Case Studies

- Redundant function call in Bwaves

```
41.  ros = q(1, ip1, jp1, kp1)
42.  us = q(2, ip1, jp1, kp1)/ros
     …
47.  mu = (mu + ((gm-1.0d0)*(q(5,ip1,jp1,kp1)/ros-
     0.5d0*(us*us+vs*vs+ws*ws)))**0.75d0)/2.0d0
```

- Reuse the value from the previous call
- Total cycles reduced by 16% for this loop
- Total instructions reduced by 17% for this loop

```
movsdq  0x8(%rdi,%r10,8), %xmm0:__mul:<no src>
  __dvd: <no src>
   __mpexp:<no src>
    __mplog:<no src>
     __slowpow:<no src>
      __ieee754_pow_sse2:<no src>
       pow:<no src>
        jacobian_:jacobian_lam.f:47
         shell_:shell_lam.f:193
          MAIN__:flow_lam.f:63
           main:flow_lam.f:67
**************REDUNDANT WITH *************
movsdq  0x8(%rdi,%r10,8), %xmm0:__mul:<no src>
 __dvd:<no src>
  __mpexp:<no src>
   __mplog:<no src>
    __slowpow:<no src>
     __ieee754_pow_sse2:<no src>
      pow:<no src>
       jacobian_:jacobian_lam.f:47
        shell_:shell_lam.f:193
         MAIN__:flow_lam.f:63
          main:flow_lam.f:67
```

# Case Studies

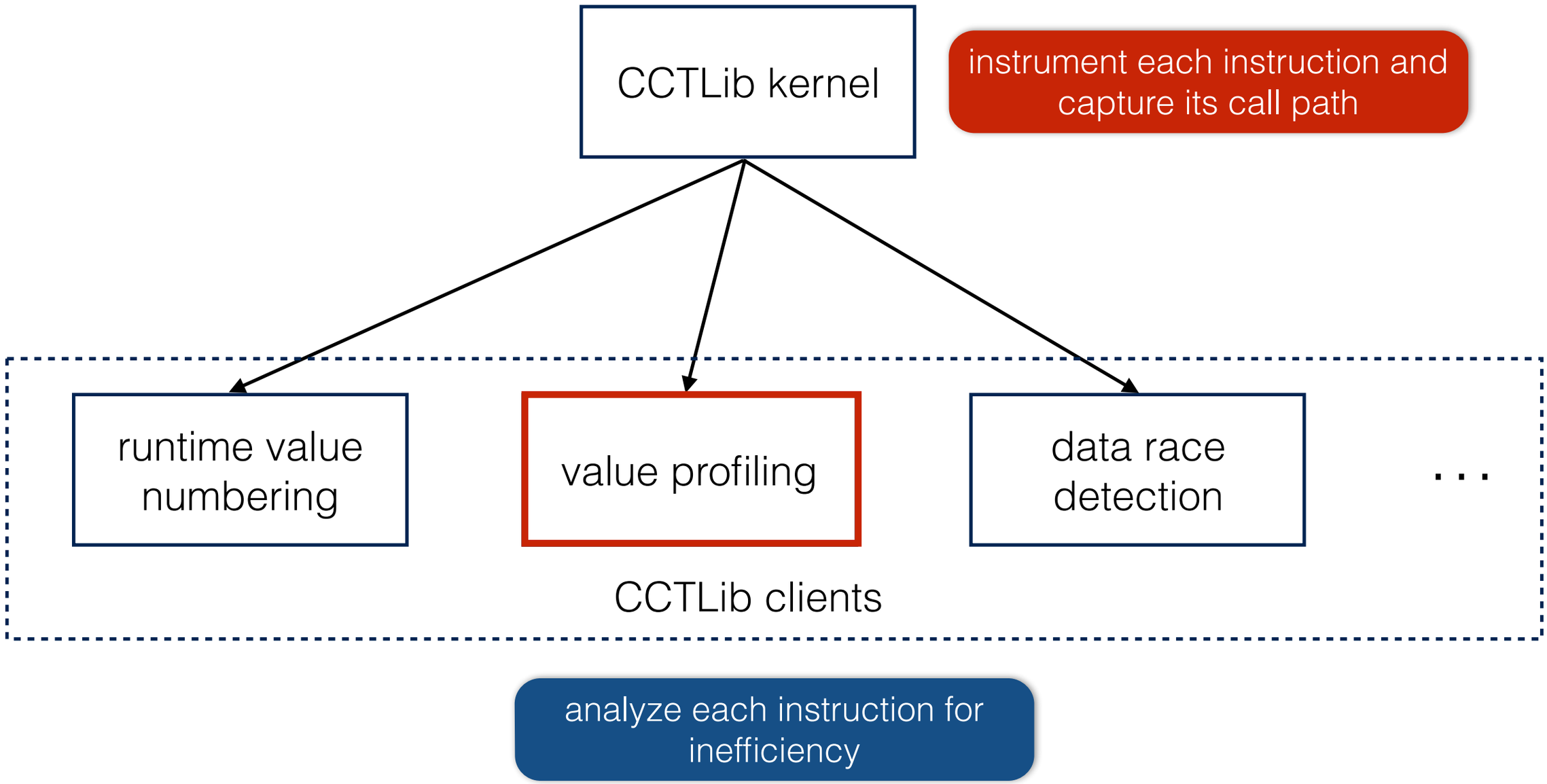
- Redundant mod in Bwaves

```
1.  do  k = 1, nz
2.      …
3.      do j = 1, ny
4.          jm1 = mod(j+ny-2, ny)+1
5.          jp1 = mod(j, ny) + 1
6.          …
7.      enddo
8.  enddo
```

```
1.  j = 1; jm1 = ny; jp1 = 2
2.  …
3.  do j = 2, ny - 1
4.      jm1 = j - 1
5.      jp1 = j + 1
6.      …
7.  enddo
8.  j = ny; jm1 = ny - 1; jp1 = 1
```

- Reduce strength of heavyweight instruction mod with lightweight ones
- Total cycles reduced by 97.1%, total instructions reduced by 97.3% for this loop

# On-going Work I



# On-going Work I: Value Profiling

- Instructions are not redundant but values are redundant

```
int TemporalRedundancy(int a, int b){
    int m = a * a;
    int n = b * b;
    int v1 = m - n;
    //v1 is used here(read)
    c = a - b;
    d = a + b;
    v1 = c * d;
    return v1;
}
```

a * a - b * b == (a + b) * (a - b)

computation for the 2nd v1 is redundant

- Value locality
  - temporal value locality
    - redundant values in the same register or memory location
  - spatial value locality
    - redundant values in adjacent memory locations

# On-going Work II: CCTLib GUI

- Fine-g
  - nee

block_solver.f

```
78                   enddo
79                enddo
80                call mat_times_vec(v,p,a,axp,ayp,azp,axm,aym,azm,
81     $          nb,nx,ny,nz)
82                tmp=0.0d0
83
84                do k=1,nz
85                   do j=1,ny
                        do i=1,nx
```

Calling Context View

| Scope | Instructions (I) |
|---|---|
| Experiment Aggregate Metrics | 3.30e+10 100 % |
| ~unknown-proc~ | 3.30e+10 100 % |
| _start | 3.30e+10 100.0 |
| 118: ~unknown-proc~ | 3.30e+10 100.0 |
| main | 3.30e+10 100.0 |
| 67: driver | 3.30e+10 100.0 |
| 63: shell | 3.30e+10 100.0 |
| loop at shell_lam.f: 159 | 3.29e+10 100.0 |
| 298: bi_cgstab_block | 1.79e+10 54.4% |
| loop at block_solver.f: 134 | 1.74e+10 52.9% |
| 81: mat_times_vec | 7.38e+09 22.4% |

serve as grand truth to quantify the accuracy of call path profilers based on PMU sampling

# Conclusions and Future Work

Fine-grained analysis is powerful

Try it now: https://github.com/chabbimilind/cctlib

# Questions?