IEEE CLUSTER 2024
Kobe, Japan • 24–27 September

Tutorial: Identifying Software Triviality via Fine-grained and Dataflow-based Value Profiling

Zhibo Xuan

Beihang University

Hands-on Tutorial @ CLUSTER24

北京航空航天大學
BEIHANG UNIVERSITY

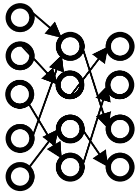
Outline

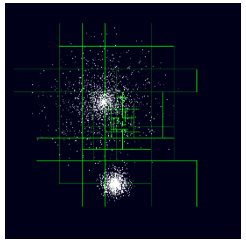
- Introduction & Background
- Understanding Software Triviality
- Dataflow-based Triviality Detection
- Evaluation
- Hands-on Tutorial
  - Installation
  - Case Study – Backprop
  - Case Study – IS Benchmark
  - Case Study – PENNANT

Outline

- Introduction & Background
- Understanding Software Triviality
- Dataflow-based Triviality Detection
- Evaluation
- Hands-on Tutorial
    - Installation
    - Case Study – Backprop, IS Benchmark
    - Case Study – LU Benchmark, BT Benchmark

From Resource Usage to Resource Wastage






Profiling

High Resource Usage (Hotspot) Analysis

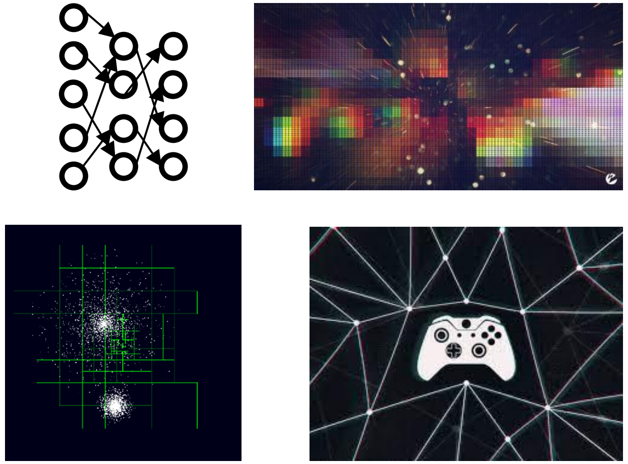
- Monitors metrics at a coarse granularity
  - Instructions per cycle (IPC), cache miss, …
  - Quantifies the average behavior over a time window
- Leads to insufficient insight for optimization

Code regions with high resource usage are likely to reveal optimization potentials, it still cannot reflect whether the resources utilized effectively.

- *log(1) in a loop is wasteful use of FPU, although it has high resource usage as well as high IPC.*

Performance analysis needs to focus on resource wastage in addition to resource usage

Resource Wastage - Trivial Operation



sophisticated control flows and deep layers of abstractions

Unexpected Software Inefficiencies

Dead Store

Silent Store

Redundant load

Redundant Zero

Trivial Operation

*e.g., 0*x, log(0), ... in loops*

Pervasively exist in real-world applications with notable negative performance impacts

- A trivial operation will always result in **the same value when specific conditions are satisfied.**
- Executing trivial operations leads to a **waste of functional units and memory bandwidth**, revealing new performance optimization opportunities.

Trivial Operation – An example

```
for(int i =0; i < N ;++ i ) { c [ i ] = i %2; D [ i ] = rand () ; }
...
for(int i =0; i < N ;++ i ) {
◁        bwd = HEAVY ( D [ i ]) ;
▶        R [ i ] = R [ i ] + sqrt ( c [ i ] × bwd ) ;
}
```

Trivial Operations

- ***Chained trivial operations*** are trivial operations that can be triggered in sequence with the same conditions
- ***Redundant backward slices*** are the dead codes when the trivial operations are eliminated with the specific conditions

Trivial Operation – An example

Chained Trivial Operations

```
for(int i =0; i < N ;++ i ) { c [ i ] = i %2; D [ i ] = rand () ; }
...
for(int i =0; i < N ;++ i ) {
◁        bwd = HEAVY ( D [ i ]) ;
▶        R [ i ] = R [ i ] + sqrt ( c [ i ] × bwd ) ;
}
```

c[i]=0? 0*bwd sqrt(0) R[i]+0

Dead Code

- ***Chained trivial operations*** are trivial operations that can be triggered in sequence with the same conditions
- ***Redundant backward slices*** are the dead codes when the trivial operations are eliminated with the specific conditions

Trivial Operation – An example

Redundant backward slices

```
for(int i =0; i < N ;++ i ) { c [ i ] = i %2; D [ i ] = rand () ; }
...
for(int i =0; i < N ;++ i ) {
◁          bwd = HEAVY ( D [ i ]) ;
▶          R [ i ] = R [ i ] + sqrt ( c [ i ] × bwd ) ;
}
```

*0*bwd = 0*

c[i]=0?

Dead Code

- ***Chained trivial operations*** are trivial operations that can be triggered in sequence with the same conditions
- ***Redundant backward slices*** are the dead codes when the trivial operations are eliminated with the specific conditions

Trivial Operation – An example

```
for(int i =0; i < N ;++ i ) { c [ i ] = i %2; D [ i ] = rand () ; }
...
for(int i =0; i < N ;++ i ) {
◁        bwd = HEAVY ( D [ i ]) ;
▶        R [ i ] = R [ i ] + sqrt ( c [ i ] × bwd ) ;
}
```

Optimzed

```
for(int i =0; i < N ;++ i ) {
    if (c[i]!=0) {
◁            bwd = HEAVY ( D [ i ]) ;
▶            R [ i ] = R [ i ] + sqrt ( c [ i ] × bwd ) ;
    }
}
```

Optimizes performance by avoiding redundant computations and memory accesses.

- ***Chained trivial operations*** are trivial operations that can be triggered in sequence with the same conditions
- ***Redundant backward slices*** are the dead codes when the trivial operations are eliminated with the specific conditions

Outline

- Introduction & Background
- Understanding Software Triviality
- Dataflow-based Triviality Detection
- Evaluation
- Hands-on Tutorial
  - Installation
  - Case Study – Backprop, IS Benchmark (OP)
  - Case Study – LU Benchmark, BT Benchmark (OP)

Software Trivialities

- ***Absorbing Triviality***
  - The triviality is absorbing if the trivial condition makes other operand irrelevant to the result when satisfied.
- ***Identical Triviality***
  - The triviality is identical if the operation result equals to other operand when the trivial condition is satisfied.
- ***Functional Triviality***
  - The triviality is functional if the trivial condition $x \equiv c$ leads to a constant operation result where $c \in \{0, 1, F\}$.

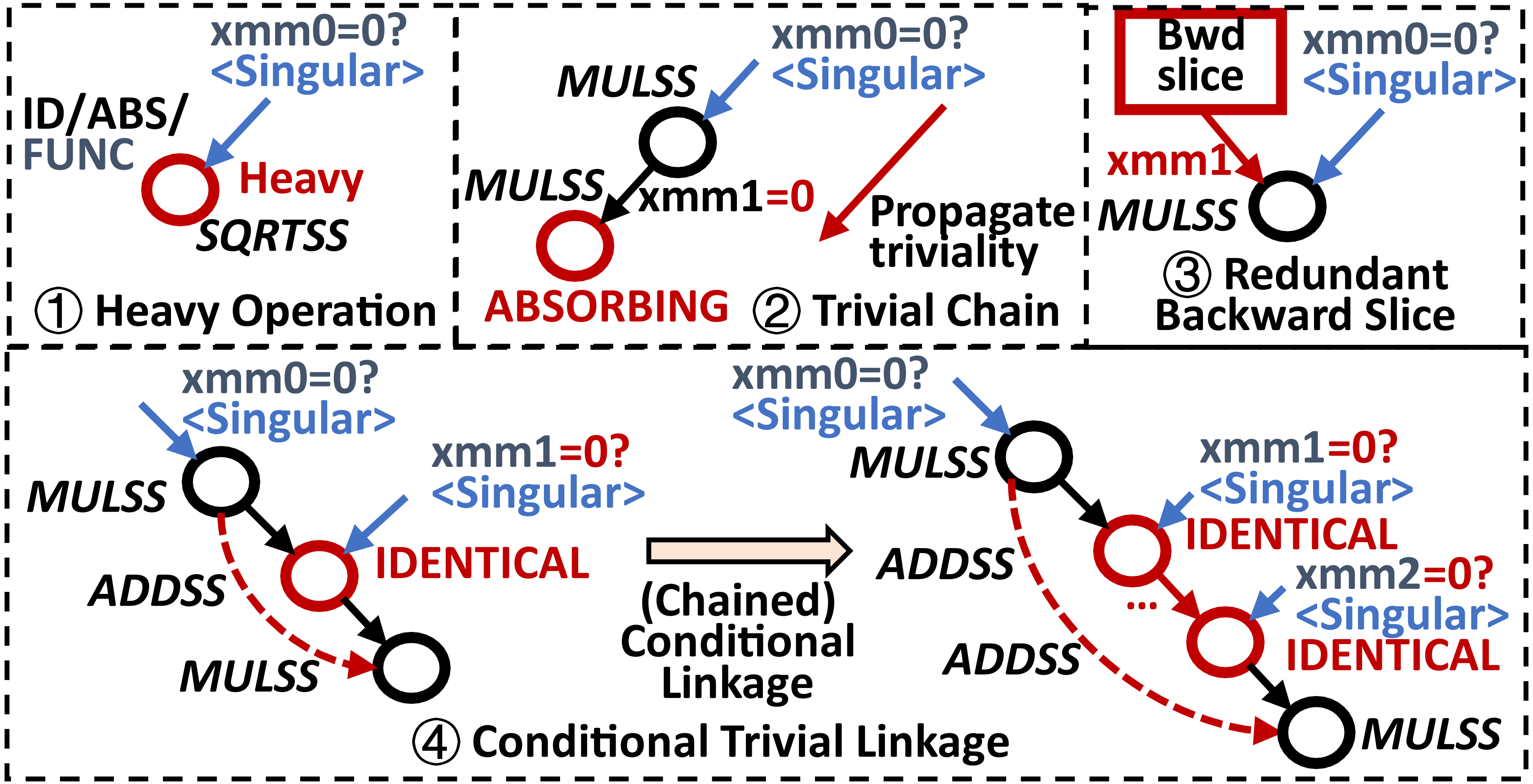
An operation is ***an instruction*** (e.g., MULSS, SQRTSS) or a ***wrapped math function*** (e.g., *exp*, *log*).

Operation	Trivial Condition	Type	Results
A & B	A=F/B=F A=0/B=0	Identical Absorbing	B/A 0
A | B	A=F/B=F A=0/B=0	Absorbing Identical	F B/A
A + B	A=0/B=0	Identical	B/A
A – B	B=0	Identical	A
A * B	A=0/B=0 A=1/B=1	Absorbing Identical	0 B/A
A / B	A=0 B=1	Absorbing Identical	0 A
sqrt(A)	A=0 A=1	Functional	0 1

* F indicates a full trivial condition, in which all bits are set to 1 with the specified data length (e.g., *0xf* for *int8*)

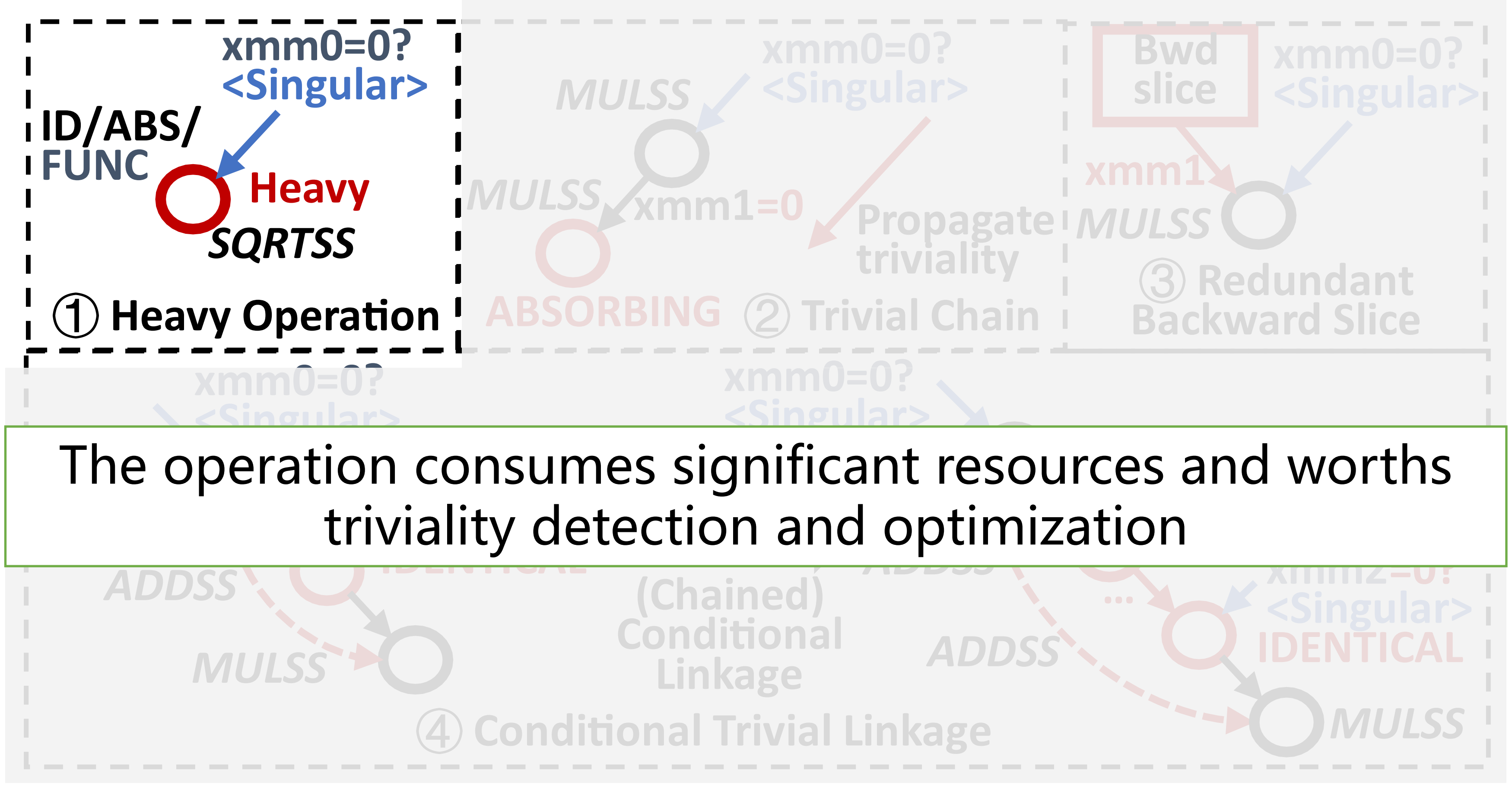
Singular Trivial Condition

- A ***Singular Trivial Condition (STC)*** is a precondition on a specific operand that triggers software trivialities with a significant performance impact.



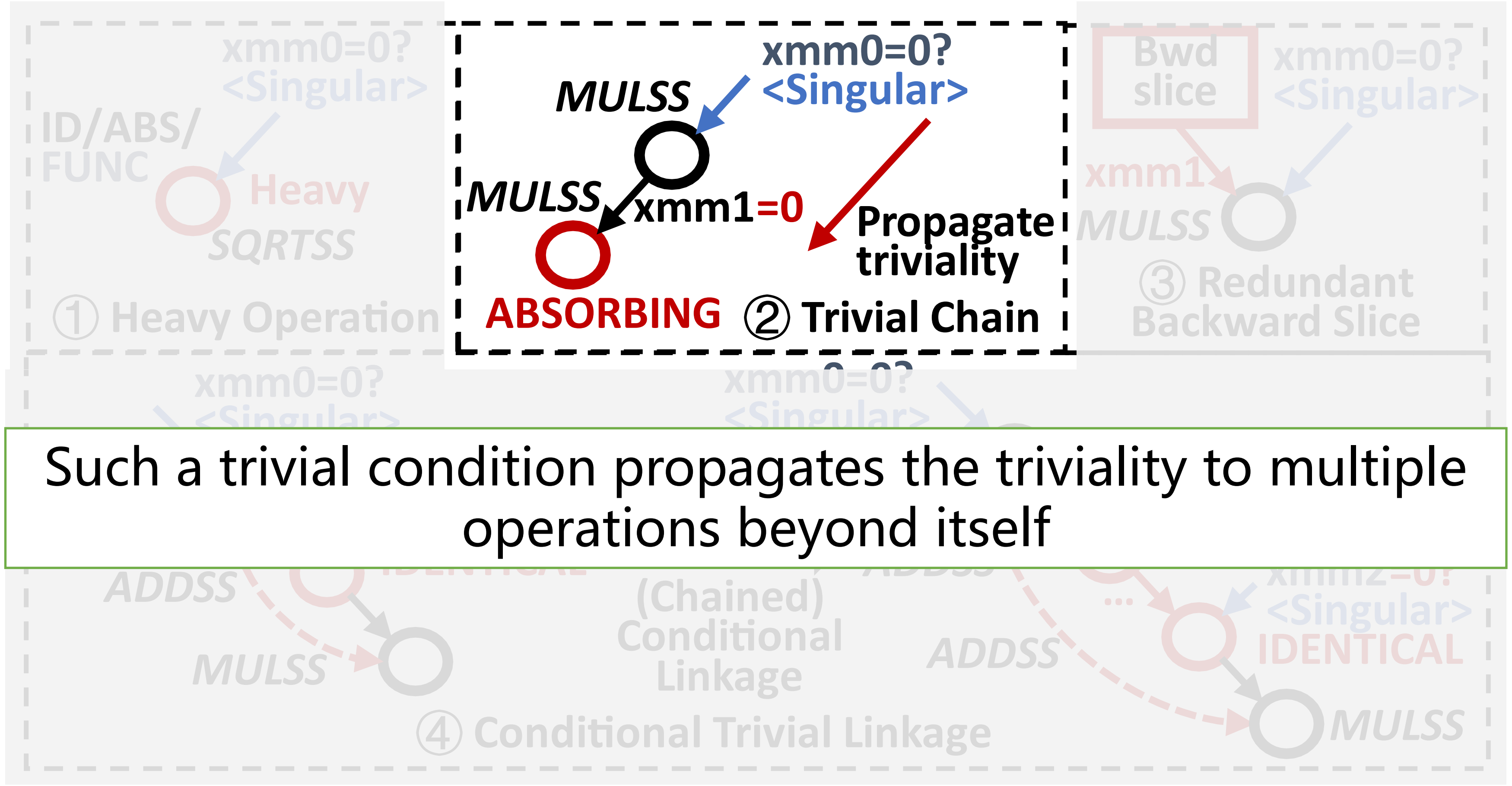
Singular Trivial Condition

- A ***Singular Trivial Condition (STC)*** is a precondition on a specific operand that triggers software trivialities with a significant performance impact.



Singular Trivial Condition

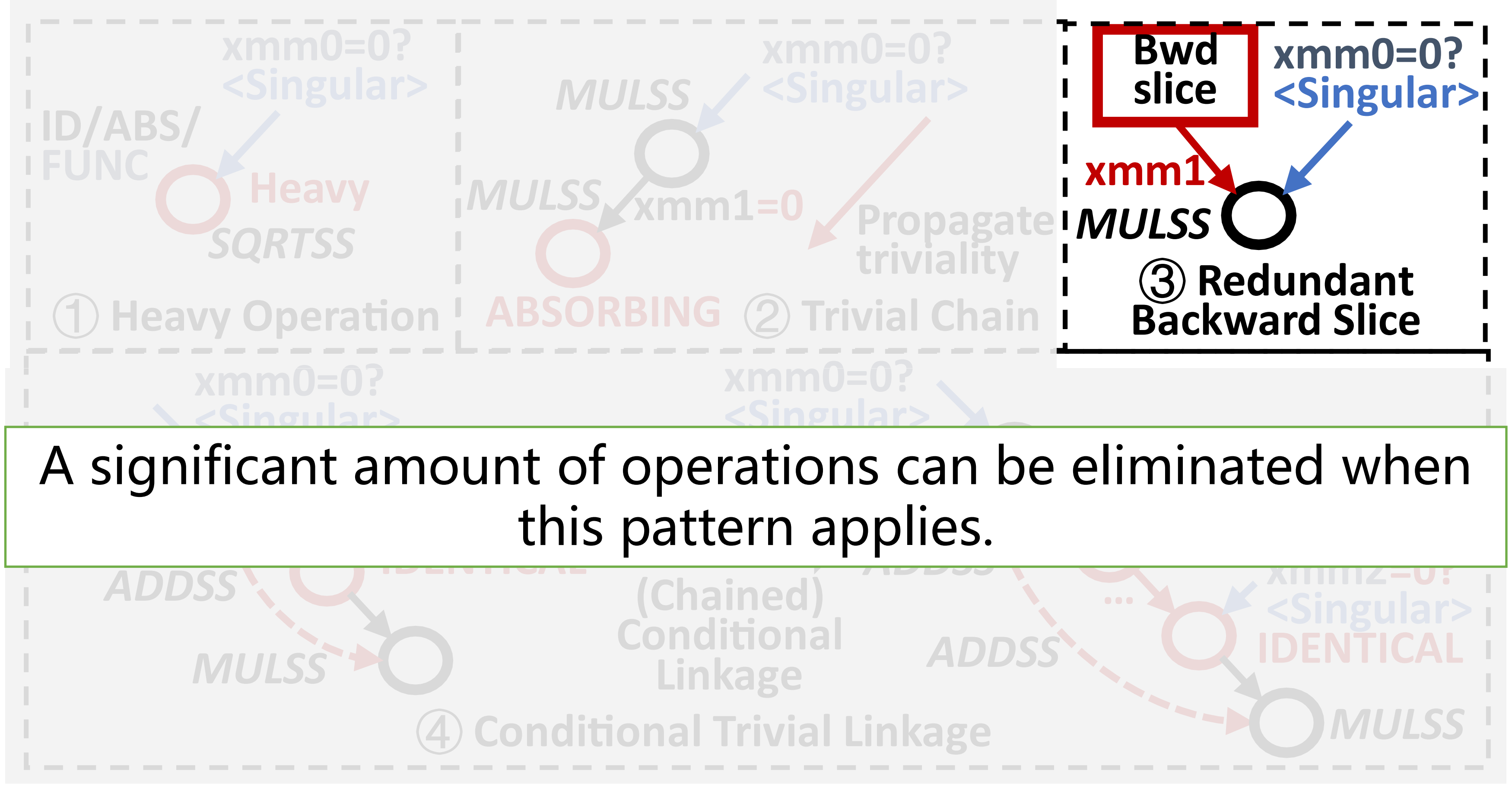
- A ***Singular Trivial Condition (STC)*** is a precondition on a specific operand that triggers software trivialities with a significant performance impact.



Such a trivial condition propagates the triviality to multiple operations beyond itself

Singular Trivial Condition

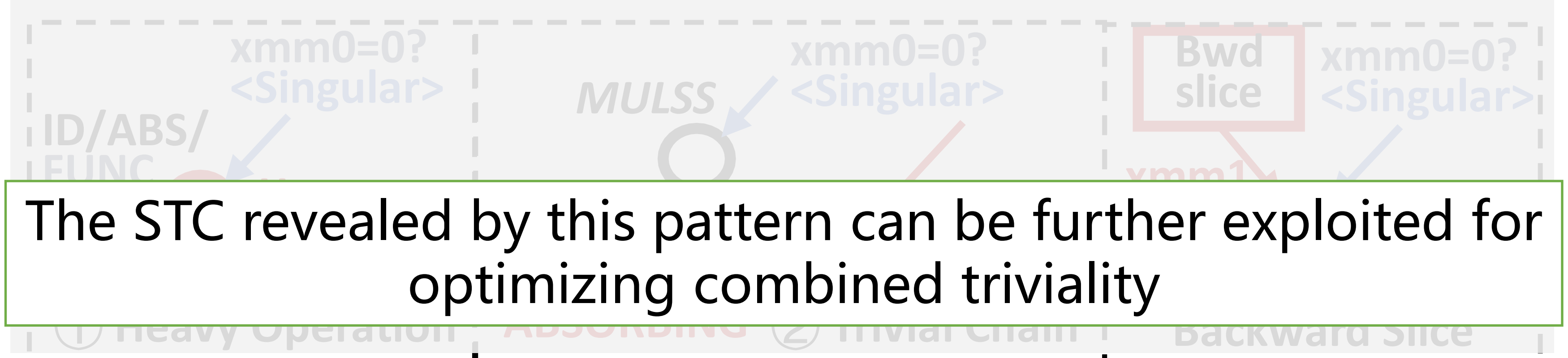
- A ***Singular Trivial Condition (STC)*** is a precondition on a specific operand that triggers software trivialities with a significant performance impact.



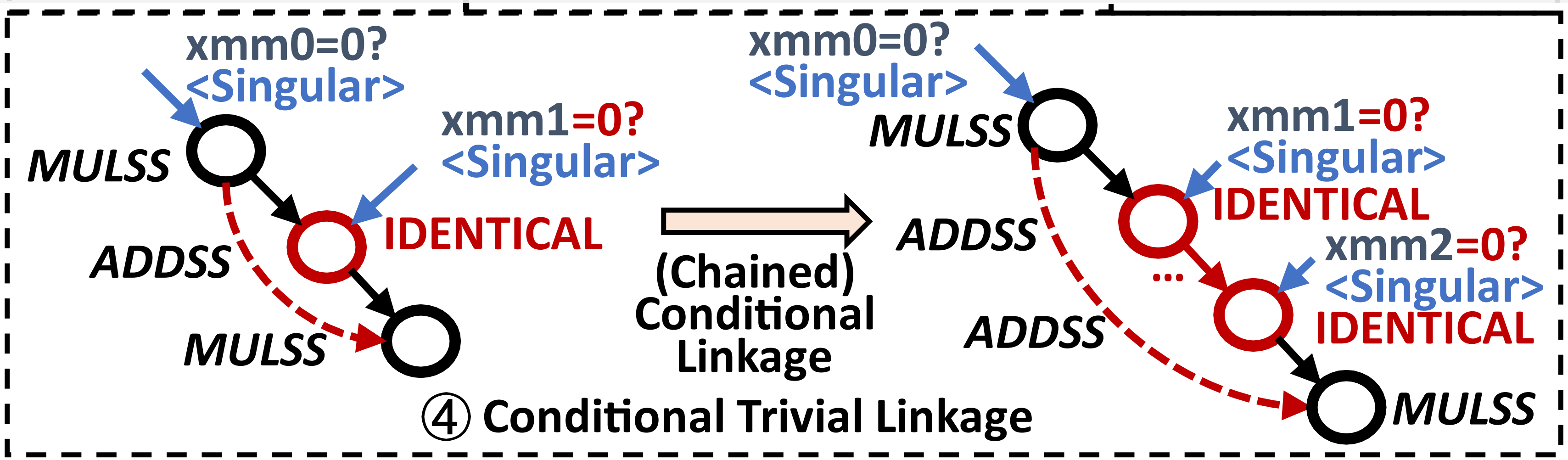
A significant amount of operations can be eliminated when this pattern applies.

Singular Trivial Condition

- A ***Singular Trivial Condition (STC)*** is a precondition on a specific operand that triggers software trivialities with a significant performance impact.



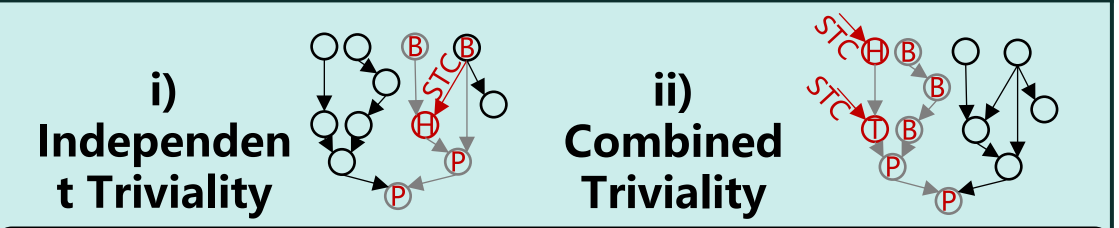
The STC revealed by this pattern can be further exploited for optimizing combined triviality



Identifying the Causes

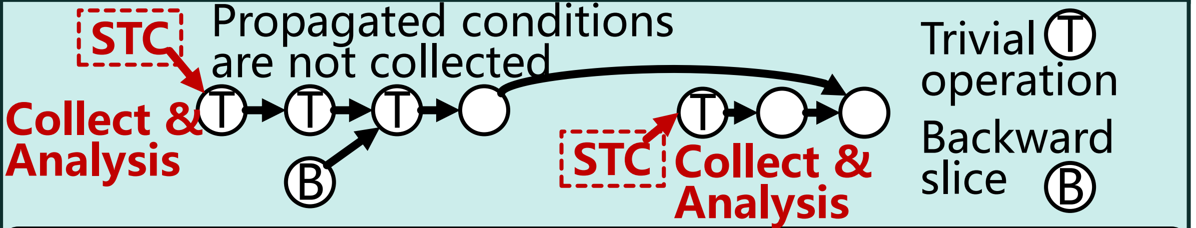
- We can derive four principles based on the above observed patterns to identify the root causes of software trivialities within acceptable overhead.

Cause Reasoning



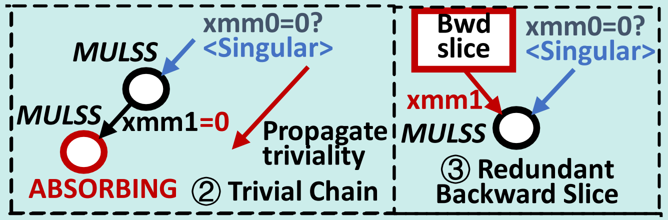
Root causes of software triviality should be easy to identify.

Detection Approach



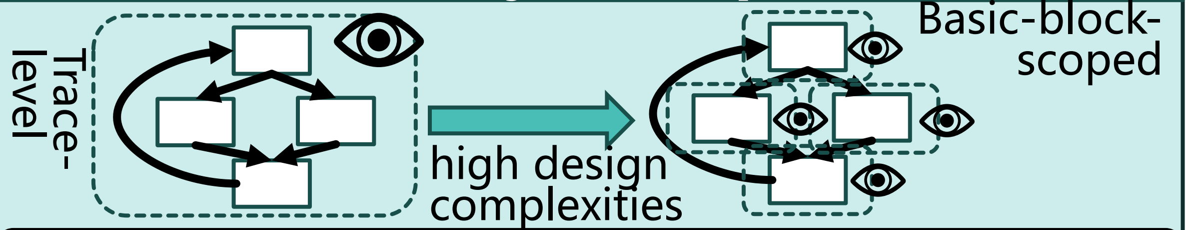
Triviality detection should be performed with reasonable overhead.

Optimization Guidance



Quantitive performance estimation is essential for actionable optimization guidance.

Analysis Scope

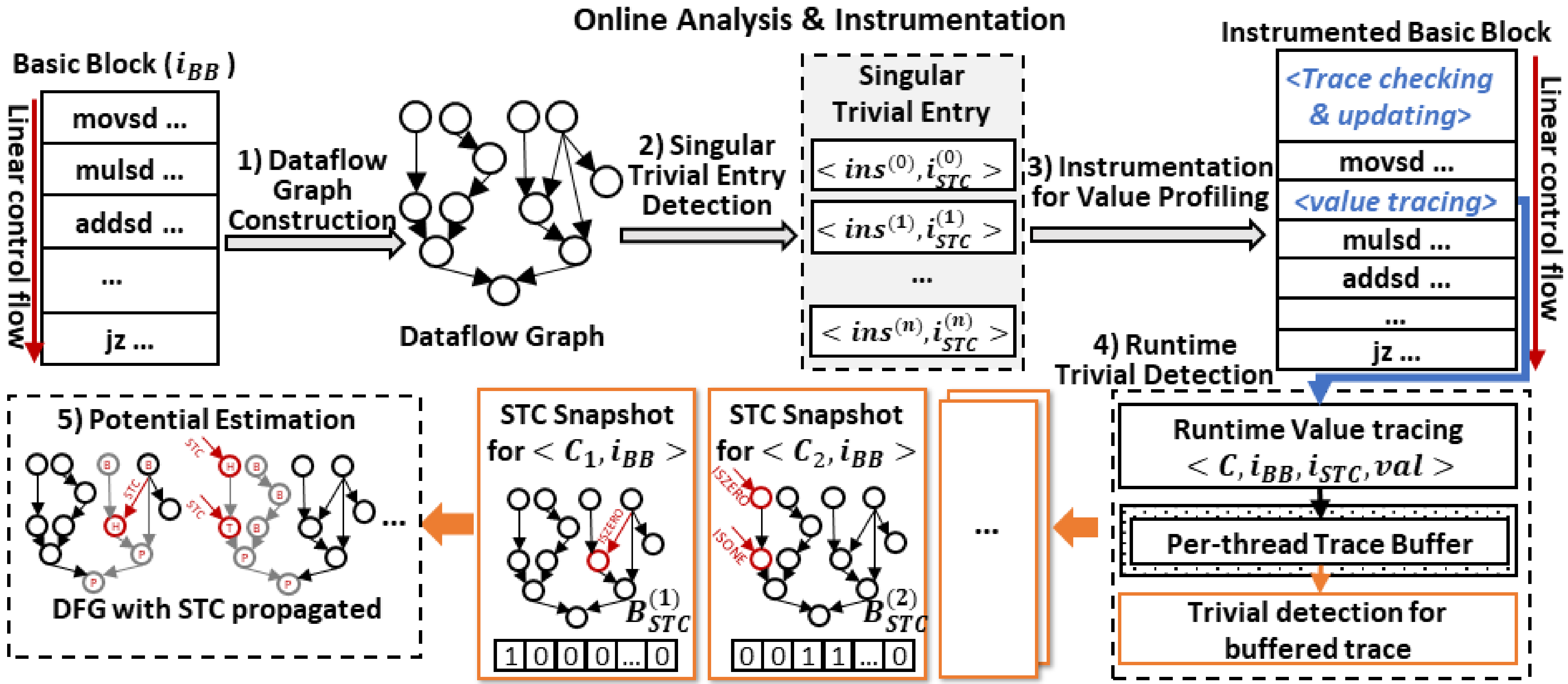


Analysis scope should balance between design complexity and profiling accuracy.

Outline

- Introduction & Background
- Understanding Software Triviality
- Dataflow-based Triviality Detection
- Evaluation
- Hands-on Tutorial
  - Installation
  - Case Study – Backprop, IS Benchmark
  - Case Study – LU Benchmark, BT Benchmark

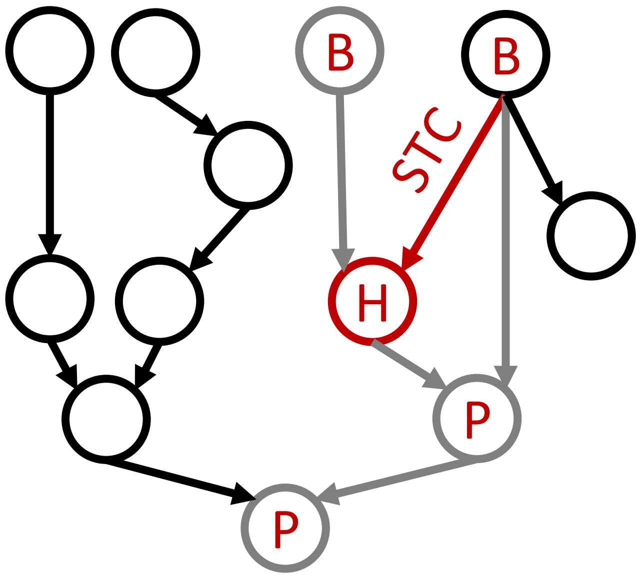
Dataflow-based Triviality Detection – Overview



- We develop a fine-grained dataflow-based value profiler *TrivialSpy* to detect and estimate the optimization potential of software triviality.

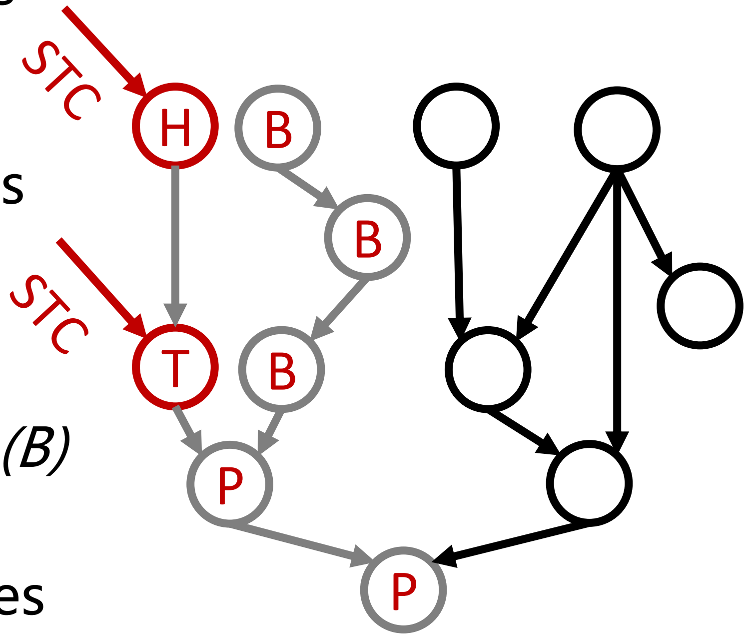
Singular Trivial Entry Detection – Independent Triviality

- ***Trivial condition selection***
  - Select first *non-propagated* trivial condition in BFS
- ***Pattern detection***
  - Applying ***Pattern 1~3***
    - *heavy operation, trivial chain, redundant backward slice*
  - Mark the nodes as *heavy (H), propagated (P), and backward (B)*
  - ***detect and mark all root causes of specific software triviality***
- ***Dead code detection***
  - all *heavy* and *propagated* nodes will be marked as dead codes
  - *backward* is considered as dead codes *iff. all its children are detected as dead codes*

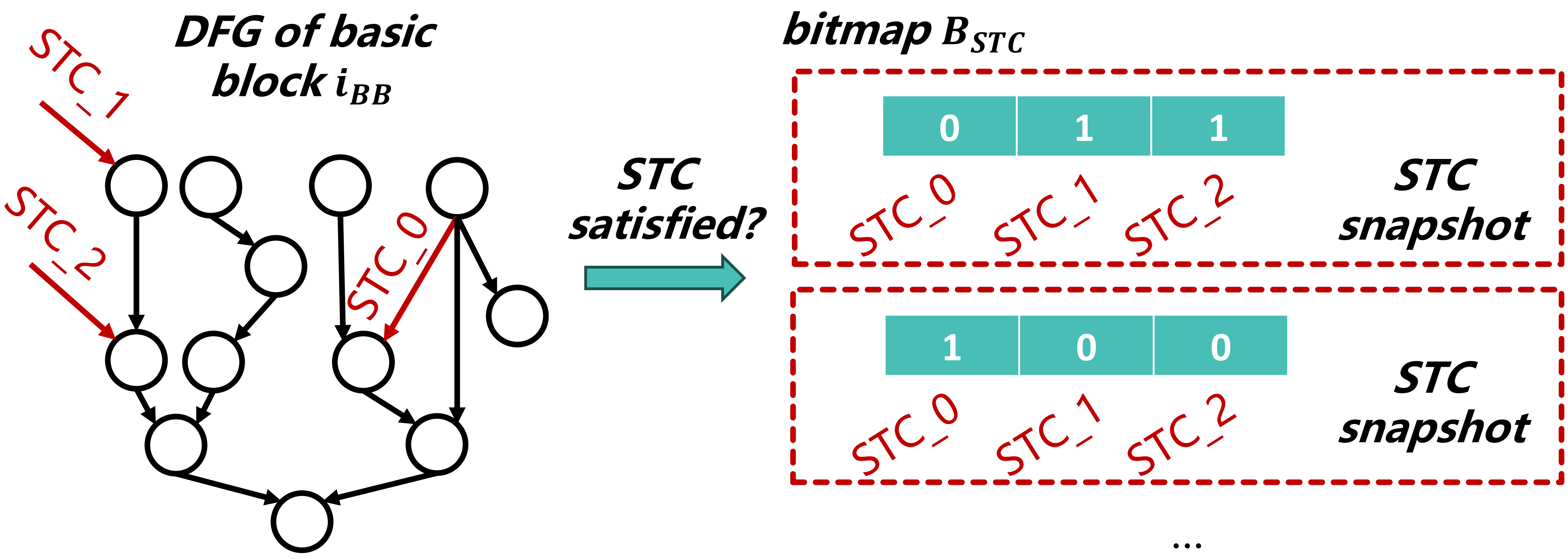


Singular Trivial Entry Detection – Combined Triviality

- ***Conditional linkage discovery***
  - Scans the nodes in the DFG with *identical triviality*
  - Each has a trivial parent generating results to make one of its children trivial
- ***Backward reasoning***
  - Analyze if the trivial conditions are propagated from previous trivial operations in the DFG
- ***Pattern detection***
  - Applying ***Pattern 1~3***
  - Mark the nodes as *heavy (H), propagated (P), and backward (B)*
- ***Dead code detection***
  - all *heavy* and *propagated* nodes will be marked as dead codes
  - *backward* is considered as dead codes *iff. all its children are detected as dead codes*



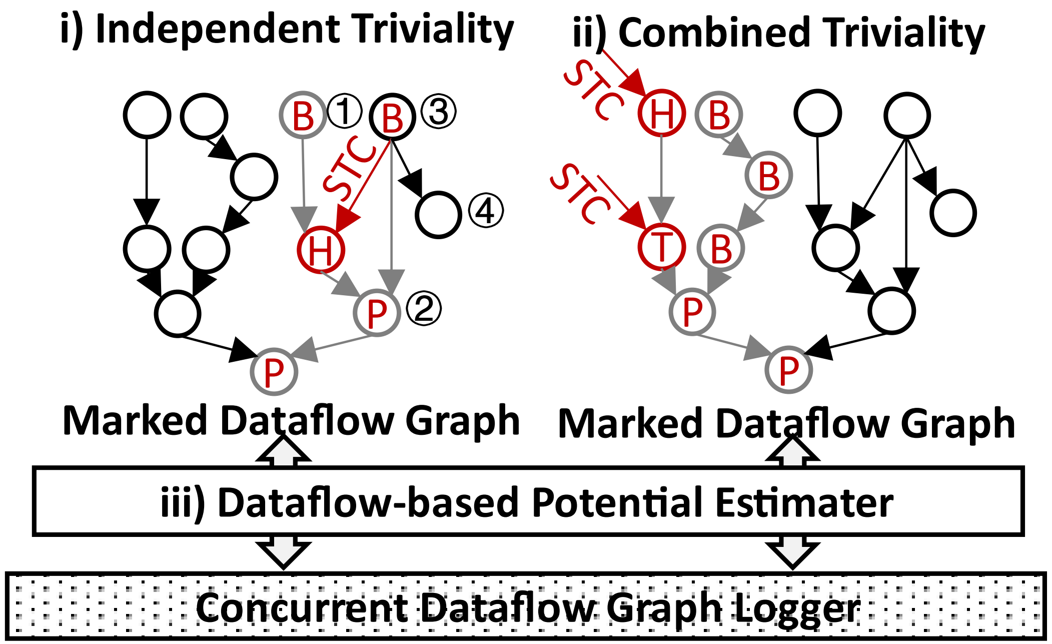
Runtime Trivial Detection



Accumulates the STC snapshots for the total number of execution with the parent’s calling context of the basic block

Potential Estimation

- **Expected Benefit *(EB)* & Branching Benefit *(BB)***
  - We mark the corresponding DFG by constantly propagating with the detected STC recorded in the snapshot (similar to *singular trivial entry detection*)
  - The performance potential of a trivial condition is estimated as the accumulated cost of operations marked as dead by the previous propagating phase



$$Cost(V) = Latency(V) + Latency_{mem} \times N_{mem}$$

$$Cost(G) = \sum_{V \in G} Cost(V)$$

$$\boxed{\boldsymbol{EB} = N_{trivial} \times \left(Cost(G) - Cost\left(G_{fast}\right)\right)}$$

$$\boxed{\boldsymbol{BB} = EB - N \times Cost(C)}$$

BB indicates the expected performance improvement of branch optimization

Reporting software trivialities

Branching Benefit: BR_{local} (BB_{local}/BB_{total}) 1
Importance: $BI_{local}(BB_{local}/Cost_{total})$

Expected Benefit: ER_{local} (EB_{local}/EB_{total}) 2
Importance: EI_{local} $(EB_{local}/Cost_{total})$

Performance potential

^^ Trivial Ratio: R_{local} $(N_{trivial}/N_{total})$ ^^ 3

+++ DFG Caller CCT Info +++ 4
<instruction>@<func>[<file>:<line>]
...

Calling context

======= DFGLog from Thread *tid* ======= 5
exe count: N
... (profiled results of $Cost(G)$, EB, HIR, TCR, $RBSR$, CTR)

Detailed metric

++++++ Singular Trivial Condition(s): 6
<i> <src, dst>: <opnd>, <STC val>, <isSingular>
...

STC

==> detailed node info: 7
[i] <instruction>@<func>[<file>:<line>] <[B][P][H]>
...
==> detailed edge info:
<i> <src, dst>: <opnd>, <propagated value>, <isSingular>
...

DFG with source lines

Outline

- Introduction & Background
- Understanding Software Triviality
- Dataflow-based Triviality Detection
- Evaluation
- Hands-on Tutorial
  - Installation
  - Case Study – Backprop, IS Benchmark
  - Case Study – LU Benchmark, BT Benchmark

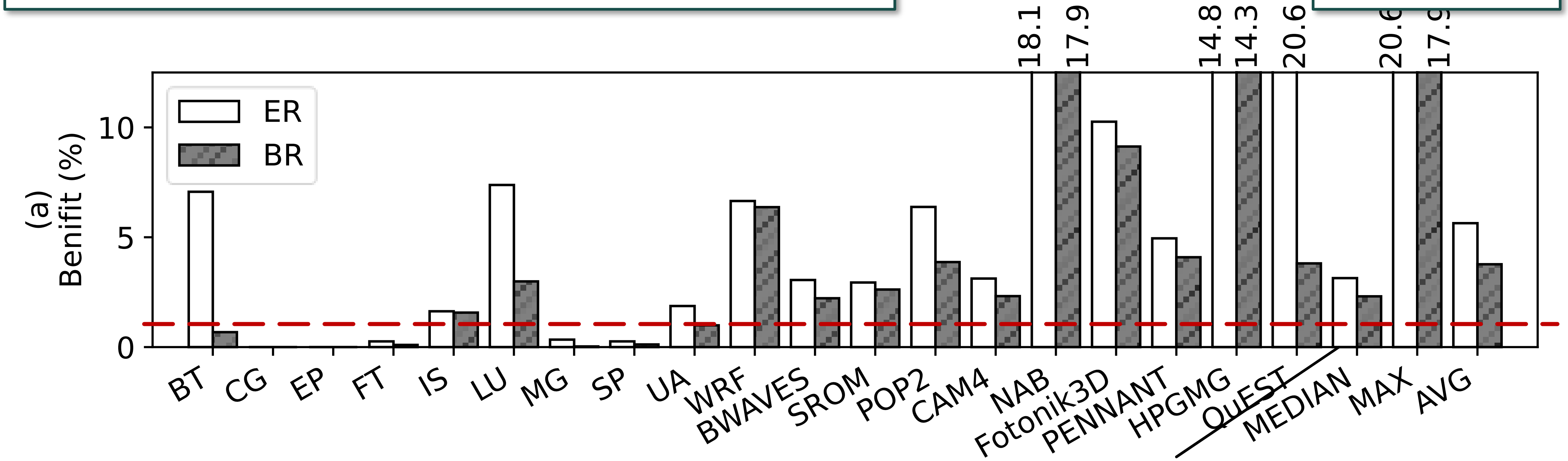
Experimental Setup

- **Server:**
  - 256 GB DDR4 memory & Intel Xeon E52680v4@2.40GHz (14 threads)
  - Linux 5.4.0-77-generic Ubuntu 20.04 LTS
- **Performance potential estimation:** $Latency_{mem} = 50$ cycles
- **Representative programs**
  - NPB 3.4.2 (Class C)
  - SPEC CPU2017 (ref)
  - CORAL2
  - QuEST
- **Compiler:** GCC 9.4.0 –O3 –fopenmp (-g for profiling)

Identifying Trivialities

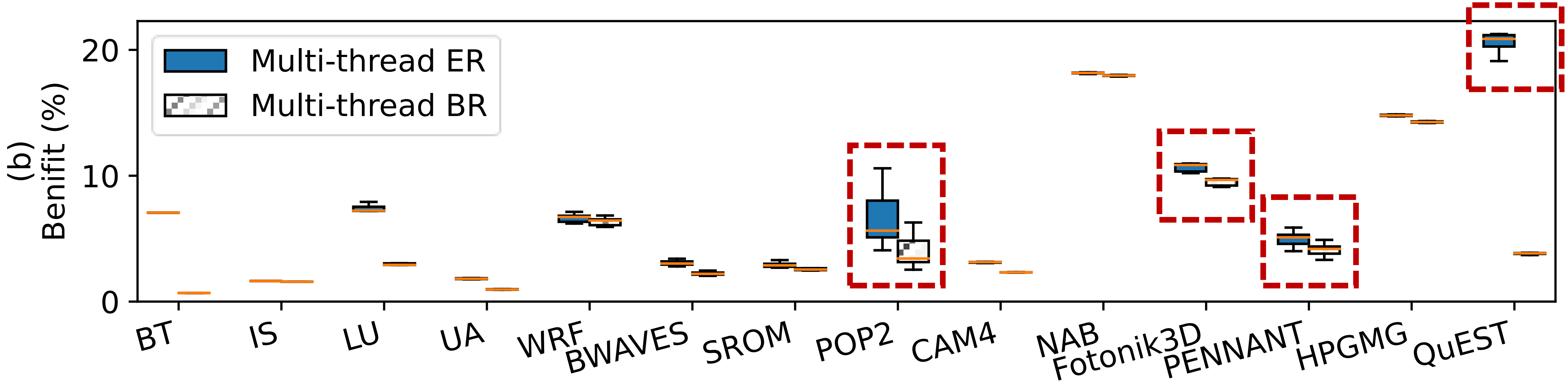
- *ER: Expected Benefit Rate*
- *BR: Branching Benefit Rate*
- ***Higher value indicates more opportunity***

AVG:
- ***ER 5.64%***
- ***BR 3.77%***



In general, we consider the ***optimization potential is actionable*** when the estimated metric ***ER or BR is larger than 1%.***

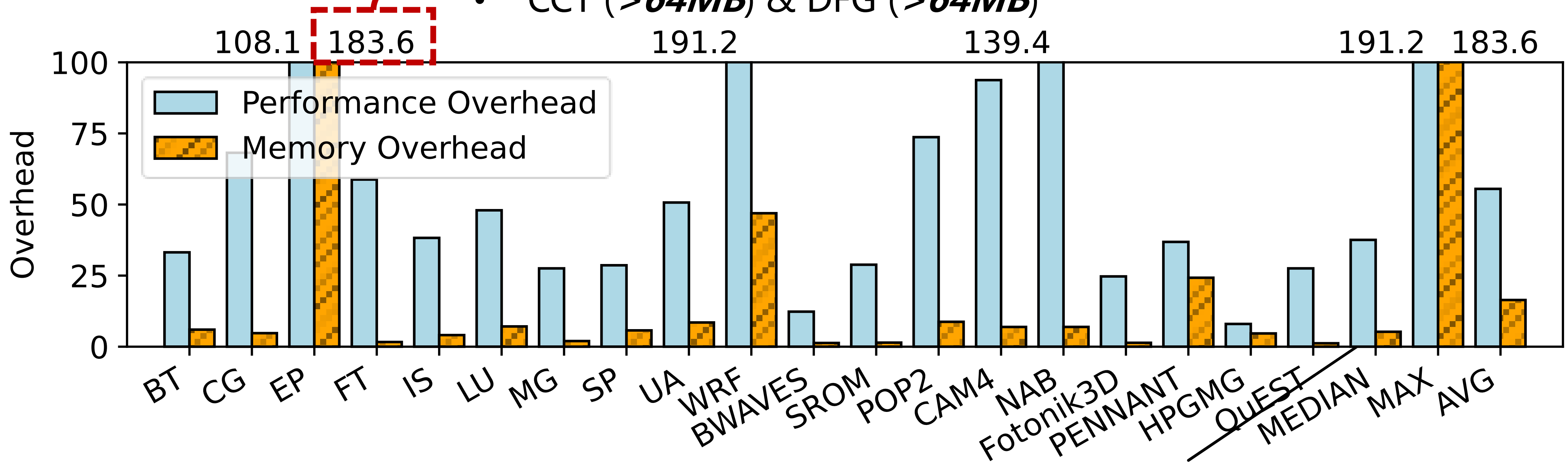
Identifying Trivialities



POP2, PENNANT, Fotonik3D and QuEST expose **large variance (≥2.38%)** of ER and BR among all threads, which indicates **potential load imbalance** caused by trivial operations.

Overhead

EP memory (***several MB***)
TrivialSpy preallocated buffer:
- CCT (***>64MB***) & DFG (***>64MB***)



The average performance and memory overhead of all evaluated programs are 55.50× and 16.62×, respectively.

The runtime overhead **is similar to well-accepted** binary instrumentation profilers.

Performance Improvement - Overview

- ***BO: branch optimization***
- ***HTFE: heavy trivial function elimination***
- ***LBO: load balance optimization***

Programs	Directed by	Code Line of Triviality	Opt.	Speedup
IS [8]	BB(*TCR*)	randlc@is.c:369	BO	4.14%±0.80%
LU [8]	BB(*RBSR,CTR*)	buts@buts.f90:loop(50-69)	BO	4.19%±0.22%
UA [8]	BB(*RBSR,CTR*)	diffusion@diffuse.f90:140	BO	2.14%±0.74%
WRF [11]	EB(*HIR*)	psim_unstable@module_sf_sfclayrev.fppized.f90:1098	HTFE	18.51%±0.21%
BWAVES [11]	BB (*HIR,RBSR,CTR*)	shell@shell_lam.fppized.f:243-270 jacobian@jacobian_lam.fppized.f:94-133	BO	1.05%±0.07%
SROM [11]	BB(*TCR,RBSR*)	pre_step3d@pre_step3d.fppized.f90:1742	BO	0.90%±0.11%
POP2 [11]	Unbalanced-EB (*HIR,CTR*)	submeso_flux@mix_submeso.fppized.f90:862 baroclinic_driver@baroclinic.fppized.f90:518	BO + LBO	2.22%±0.21%
CAM4 [11]	BB(*HIR,CTR*)	cosp_precip_mxratio@cosp_utils.fppized.f90:76	BO	1.47%±0.16%
NAB [11]	EB(*HIR*)	egb@eff.c:2107	HTFE	9.80%±0.16%
Fotonik3D [11]	BB(*CTR*)	updateh@update.fppized.f90:loop(189-201)	BO	51.11%±0.3%
	Unbalanced-EB	updateh@update.fppized.f90:189	+ LBO	52.09%±0.03%
PENNANT [1]	Unbalanced-EB (*RBSR*)	QCS::setQCnForce@QCS.cc:270 Hydro::doCycle@Hydro.cc:211	BO + LBO	2.57% ±0.06%
HPGMG [1]	BB(*RBSR,CTR*)	rebuild_operator_blackbox@rebuild.c:130	BO	1.30%±0.40%
QuEST [26]	BB(*CTR*)	statevec_controlledCompactUnitaryLocal@QuEST_cpu.c:loop(2241-2257)	BO	14.22%±0.27%
	Unbalanced-EB	statevec_controlledCompactUnitaryLocal@QuEST_cpu.c:2230	+ LBO	23.96%±0.17%

Programs with **high optimization potentials** can **obtain performance improvement** after optimization.

For more details, please refer to our paper and API docs.

X. You, H. Yang, K. Lei, Z. Luan and D. Qian, "TrivialSpy: Identifying Software Triviality via Fine-grained and Dataflow-based Value Profiling," SC23: International Conference for High Performance Computing, Networking, Storage and Analysis, Denver, CO, USA, 2023, pp. 1-14.

TrivialSpy is open-source: https://github.com/VClinic/VClinic

Outline

- Introduction & Background
- Understanding Software Triviality
- Dataflow-based Triviality Detection
- Evaluation
- Hands-on Tutorial
    - Installation
    - Case Study – Backprop, IS Benchmark
    - Case Study – PENNANT, BT Benchmark

Installation - TrivialSpy

- **Install with source code:**
  - Dependencies: git, gcc>=9, g++, make, cmake>=3.20
  - Source Code: `git clone https://github.com/VClinic/VClinic`
  - Compilation Instruction: `cd VClinic && ./build.sh`
  - Configure the Path: ``export DRRUN=`pwd`/build/bin64/drrun``
- **Install in tutorial cluster:**
  - Source Code: `cp –r /public/home/buaa_hipo/shared_folder/VClinic ./`
  - Compilation Instruction: `cd VClinic && ./build.sh`
  - Configure the Path: ``export DRRUN=`pwd`/build/bin64/drrun``
- **Instruction to analyze the target program:**
  - TrivialSpy: `$DRRUN –t trivialspy -- <EXE> <ARGS>`

Case Study – backprop (~6mins)

- Get the Benchmark and Compile： backprop
  - Get the rodinia_3.1：

```
cp –r /public/home/buaa_hipo/shared_folder/backprop ./
```

  - Compile：

```
cd /backprop
vim Makefile  # add the –g to all flags
make
cp slurm-template.sh trivial-backprop.sh
vim trivial-backprop.sh
sh trivial-backprop.sh
```

Modify the Job Name

Modify the EXE PATH

```
NTASKS=1
#!/bin/bash
JOBNAME="vclinic-trivialspy-backprop"
DRRUN="`pwd`/VClinic/build/bin64/drrun"
TOOL="trivialspy"
# TARGET command for profiling: CMD="<EXE> <ARGS>"
CMD="`pwd`/backprop "
mkdir -p log

echo "START $JOBNAME WITH NTASK=$NTASKS "
nowdate=$(date +%Y_%m_%d_%H_%M_%S)
echo $nowdate
sbatch << END
#!/bin/bash
#SBATCH -J $JOBNAME
#SBATCH -o log/$JOBNAME-$NTASKS-%j-$nowdate.log
#SBATCH -e log/$JOBNAME-$NTASKS-%j-$nowdate.err
#SBATCH -p test
#SBATCH --cpus-per-task=16
#SBATCH --ntasks-per-node=1
#SBATCH -n $NTASKS
# Your SCRIPT commands
time $CMD 6553600
time $DRRUN -t $TOOL -- $CMD 65536
END
```

Unoptimized time

Profiling time

Case Study – backprop

- Resulting files are generated in the x86-<host>-<PID>-trivialspy folder
  - trivialspy.log is the summary report for invalid operation detection metrics
  - thread-<id>.log contains the invalid operation detection reports for individual threads.
  - Summary reports: backprop has an expected benifit of 1.300 and a branch benifit of 0.866.

```
Running: /public/home/csjt0800/xzb/VClinic/build/bin64/../clients/lib64/release/libtrivialspy.so
[TRIVIALSPY INFO] Profiling with value tracing
[TRIVIALSPY INFO] Thread Private is disabled.
[TRIVIALSPY INFO] Soft Approximation is disabled
[TRIVIALSPY INFO] Hard Approximation is disabled

------ [Thread=1] Dumping Dataflow-aware Trivial Inefficiency Overview Report ------

Total Speculate Benifit: 1.300 (1570475 benifit / 120837540 total cost)
Total Benifit: 0.866 (1046821 benifit / 120837540 total cost)
Total Heavy Instruction: 0.010 (153 / 1570475 SB)
Total Trivial Chain: 49.996 (785176 / 1570475 SB)
Total Redundant Backward Slice: 0.004 (58 / 1570475 SB)
Total Absorbing Breakpoints: 99.979 (1570140 / 1570475 SB)
```

trivialspy.log

Case Study – backprop

- Optimization Guide： the xmm0, xmm1 of backprop.c:369 leading to chained trivial operation

thread-1.log

```
+++ DFG Summary Info +++
======= DFGLog from Thread 2761 =======
exe count: 130024
total cost: 433
benifit: 12
TC branch cost: 4
heavy cost: 0
chained cost: 6
backward slice cost: 0
Trivial Condition Num: 2

++++++ Singular Trivial Condition(s):
  <8>   <4, 5>: opnd=xmm0, val=ZERO, isSingular=1
  <20>  <11, 12>: opnd=xmm1, val=ZERO, isSingular=1
==> detailed node info:
  [4]   cvtss2sd xmm0, …… backprop.c:323]
  [5]   mulsd  xmm0, xmm2 …… backprop.c:323] [D]
  [6]   cvtss2sd xmm1, dword ptr [r9+rax*4] …… backprop.c:323] [B]
  [10]  pxor   xmm1, xmm1 …… backprop.c:323]
  [11]  cvtss2sd xmm1, dword ptr [rdx] …… backprop.c:323]
  [12]  mulsd  xmm1, xmm2 …… backprop.c:323] [D]
  [13]  addsd  xmm0, xmm1 …… backprop.c:323] [D][P]
==> detailed edge info:
  <4>   <-1, 2>: opnd=xmm1, val=UNKNOWN, isSingular=0
  <5>   <-1, 3>: opnd=qword ptr [r10+rax*8], val=UNKNOWN, isSingular=0
  <11>  <1, 7>: opnd=rdx, val=UNKNOWN, isSingular=0
  <12>  <-1, 8>: opnd=rsi, val=UNKNOWN, isSingular=0
```

backprop.c

```
#ifdef OPEN
  omp_set_num_threads(NUM_THREAD);
  #pragma omp parallel for  \
      shared(oldw, w, delta) \
          private(j, k, new_dw) \
          firstprivate(ndelta, nly)
#endif
  for (j = 1; j <= ndelta; j++) {
    for (k = 0; k <= nly; k++) {
      new_dw = ((ETA * delta[j] * ly[k]) + (MOMENTUM * oldw[k][j]));
          w[k][j] += new_dw;
          oldw[k][j] = new_dw;
    }
  }
}
```

In many cases, both **delta[j]** and **oldw[k][j] will be zero**, resulting in a large number of trivial operations for updating **new_dw, w[k][j], and oldw[k][j]**.

new_dw = 0 if delta[j]==0 && oldw[k][j]==0.

Case Study – backprop

- Optimization Guide： add the branch code.

backprop.c

```
#ifdef OPEN
  omp_set_num_threads(NUM_THREAD);
  #pragma omp parallel for  \
      shared(oldw, w, delta) \
          private(j, k, new_dw) \
          firstprivate(ndelta, nly)
#endif
  for (j = 1; j <= ndelta; j++) {
    for (k = 0; k <= nly; k++) {
      new_dw = ((ETA * delta[j] * ly[k]) + (MOMENTUM * oldw[k][j]));
      w[k][j] += new_dw;
      oldw[k][j] = new_dw;
    }
  }
}
```

In many cases, both **delta[j]** and **oldw[k][j] will be zero**, resulting in a large number of trivial operations for **new_dw**.

new_dw = 0 if delta[j]==0 && oldw[k][j]==0.

```
vim backprop/backprop.c
```

backprop.c

```
#ifdef OPEN
  omp_set_num_threads(NUM_THREAD);
  #pragma omp parallel for  \
      shared(oldw, w, delta) \
          private(j, k, new_dw) \
          firstprivate(ndelta, nly)
#endif
  for (j = 1; j <= ndelta; j++) {
    for (k = 0; k <= nly; k++) {
      if (delta[j]==0 && oldw[k][j]==0){
        new_dw=0;
      }
      else{
        new_dw = ((ETA * delta[j] * ly[k]) + (MOMENTUM * oldw[k][j]));
        w[k][j] += new_dw;
        oldw[k][j] = new_dw;
      }
    }
  }
}
```

```
make  #recompilation
```

Improve 20.6% performance

Case Study – IS Benchmark (~6mins, Optional)

- Get the Benchmark and Compile： IS - Integer Sort, random memory access
  - Get the NPB Benchmark：

```
cp –r /public/home/buaa_hipo/shared_folder/NPB-3.4.2-OMP ./
```

  - Compile：

```
cd NPB3.4-OMP/
cp config/make.def.template config/make.def
vim config/make.def # add the –g to all flags
make IS CLASS=C && cd ..
cp slurm-template.sh trivial-is.sh
vim trivial-is.sh && sh trivial-is.sh
```

```
#---------------------------------------------------------------
# Global *compile time* flags for Fortran programs
#---------------------------------------------------------------
FFLAGS   = -O3 -fopenmp -g
```

```
# _FILE_OFFSET_BITS=64
# _LARGEFILE64_SOURCE - are standard compiler flags which
# files larger than 2GB.
#---------------------------------------------------------------
CFLAGS   = -O3 -fopenmp -g
```

Modify the Job Name

Modify the EXE PATH

```
NTASKS=1
#!/bin/bash
JOBNAME="vclinic-trivialspy-is"
DRRUN="`pwd`/VClinic/build/bin64/drrun"
TOOL="trivialspy"
# TARGET command for profiling: CMD="<EXE> <ARGS>"
CMD="`pwd`/NPB3.4-OMP/bin/is.C.x"
mkdir -p log

echo "START $JOBNAME WITH NTASK=$NTASKS "
nowdate=$(date +%Y_%m_%d_%H_%M_%S)
echo $nowdate
sbatch << END
```

```
#!/bin/bash
#SBATCH -J $JOBNAME
#SBATCH -o log/$JOBNAME-$NTASKS-%j-$nowdate.log
#SBATCH -e log/$JOBNAME-$NTASKS-%j-$nowdate.err
#SBATCH -p test
#SBATCH --cpus-per-task=16
#SBATCH --ntasks-per-node=1
#SBATCH -n $NTASKS
# Your SCRIPT commands
cd `pwd`/PENNANT/build
time -- $CMD
time $DRRUN -t $TOOL -- $CMD
END
```

Unoptimized time

Unoptimized time

Case Study – IS Benchmark (Optional)

- Analyze the IS with TrivialSpy: `$DRRUN -t trivialspy -- ./bin/is.C.x`
- Resulting files are generated in the x86-<host>-<PID>-trivialspy folder
  - trivialspy.log is the summary report for invalid operation detection metrics
  - thread-<id>.log contains the invalid operation detection reports for individual threads.
  - Summary reports: IS has an expected benifit of 25.003 speculate benifit.

trivialspy.log

=== Overall Triviality Metric ===

------ [Thread=2] Dumping Dataflow-aware Trivial Inefficiency Overview Report ------

Total Speculate Benifit: 0.089 (74528504 benifit / 84005563936 total cost)
Total Benifit: 0.030 (24842849 benifit / 84005563936 total cost)
Total Heavy Instruction: 0.000 (102 / 74528504 SB)
Total Trivial Chain: 0.000 (217 / 74528504 SB)
Total Redundant Backward Slice: 0.000 (279 / 74528504 SB)
Total Absorbing Breakpoints: 0.001 (402 / 74528504 SB)

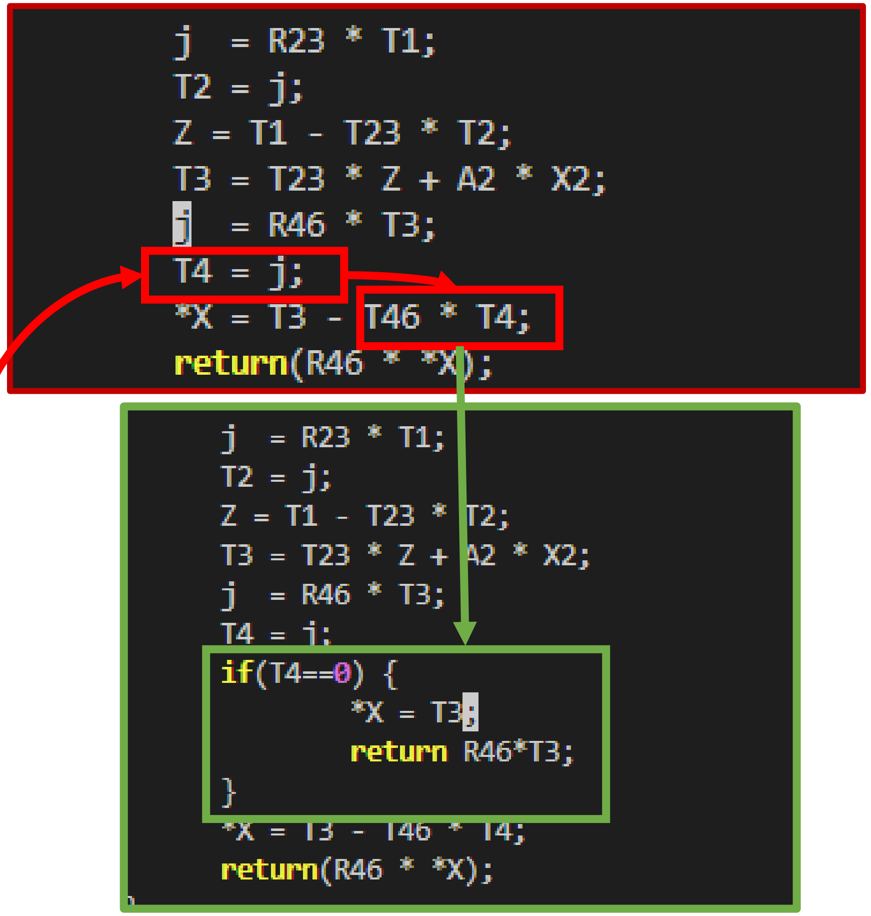
```
------ [Thread=2] Dumping Dataflow-aware Trivial Inefficiency Report ------

------- Trivial Hotspots ordered by BB --------

Total Speculate Benifit: 0.089 (74528504 benifit / 84005563936 total cost)
Total Benifit: 0.030 (24842849 benifit / 84005563936 total cost)
Total Heavy Instruction: 0.000 (102 / 74528504 SB)
Total Trivial Chain: 0.000 (217 / 74528504 SB)
Total Redundant Backward Slice: 0.000 (279 / 74528504 SB)
Total Absorbing Breakpoints: 0.001 (402 / 74528504 SB)

-------------------------------
Benifit: 25.003 (6211538 local benifit / 24842849 total benifit)
Importance: 0.007 (6211538 benifit / 84005563936 total cost)

Speculate Benifit: 25.003 (18634614 local SB / 74528504 SB)
Importance: 0.022 (18634614 benifit / 84005563936 total cost)
```

Case Study – IS Benchmark (Optional)

- Optimization Guide: the 0 value xmm3 of is.c:369 leading to chained trivial operation
  - Eliminate the propagation of invalid calculations caused by zero values resulting from converting.

```
+++ DFG Summary Info +++
======= DFGLog from Thread 31647 =======
exe count: 6210686
total cost: 278
benifit: 3
TC branch cost: 2
heavy cost: 0
chained cost: 0
backward slice cost: 0
Trivial Condition Num: 1
++++++ Singular Trivial Condition(s):
  <55>  <34, 35>: opnd=xmm3, val=ZERO, isSingular=1
==> detailed node info:
  [33]  pxor   xmm3, xmm3
@randlc[/public/home/buaa_hipo/app/NPB3.4.2/NPB3.4-OMP/IS/is.c:368]
  [34]  cvtsi2sd xmm3, eax
@randlc[/public/home/buaa_hipo/app/NPB3.4.2/NPB3.4-OMP/IS/is.c:368]
  [35]  mulsd  xmm1, xmm3
@randlc[/public/home/buaa_hipo/app/NPB3.4.2/NPB3.4-OMP/IS/is.c:369] [D]
  [36]  subsd  xmm2, xmm1
@randlc[/public/home/buaa_hipo/app/NPB3.4.2/NPB3.4-OMP/IS/is.c:369] [P]
…
```

vim NPB3.4-OMP/IS/is.c



Improve 5% performance

Case Study – fotonik3d

- Get the Benchmark and Compile： spec cpu2017/fotonik3d
  - Get the fotonik3d：

```
cp –r /public/home/buaa_hipo/shared_folder/649.fotonik3d ./
```

  - Compile：

```
cd 649.fotonik3d
make && cd ..
cp slurm-template.sh trivial-fotonik.sh
vim trivial-fotonik.sh
sh trivial-fotonik.sh
```

Modify the Job Name

Modify the EXE PATH

```
NTASKS=1
#!/bin/bash
JOBNAME="vclinic-trivialspy-fotonik"
DRRUN="`pwd`/VClinic/build/bin64/drrun"
# tool name example: zerospy, trivialspy
TOOL="trivialspy"
# TARGET command for profiling: CMD="<EXE> <ARGS>"
CMD="fotonik3d_s_base.mytest-m64"
mkdir -p log
echo "START $JOBNAME WITH NTASK=$NTASKS "
nowdate=$(date +%Y_%m_%d_%H_%M_%S)
echo $nowdate
sbatch << END
#!/bin/bash
#SBATCH -J $JOBNAME
#SBATCH -o log/$JOBNAME-$NTASKS-%j-$nowdate.log
#SBATCH -e log/$JOBNAME-$NTASKS-%j-$nowdate.err
#SBATCH -p test
#SBATCH --cpus-per-task=16
#SBATCH --ntasks-per-node=1
#SBATCH -n $NTASKS
# Your SCRIPT commands
cd 649.fotonik3d
time $CMD
time $DRRUN -t $TOOL -- $CMD
END
```

Unoptimized time

Profiling time

Case Study – fotonik3d

```
======= DFGLog from Thread 16547 =======
exe count: 8363306
total cost: 3499
benifit: 1425
TC branch cost: 180
heavy cost: 960
chained cost: 312
backward slice cost: 312
Trivial Condition Num: 15
++++++ Singular Trivial Condition(s):
  <0>    <-1, 0>: opnd=qword ptr [r15+rax*8], val=ZERO, isSingular=1
  <9>    <6, 7>: opnd=xmm2, val=ZERO, isSingular=1
  <11>   <7, 8>: opnd=xmm2, val=ZERO, isSingular=1
  <14>   <8, 9>: opnd=xmm1, val=ZERO, isSingular=1
  <28>   <16, 17>: opnd=xmm1, val=ZERO, isSingular=1
  <38>   <-1, 23>: opnd=qword ptr [rbx+rax*8], val=ZERO, isSingular=1
  <46>   <28, 29>: opnd=xmm2, val=ZERO, isSingular=1
  <48>   <29, 30>: opnd=xmm2, val=ZERO, isSingular=1
  <51>   <30, 31>: opnd=xmm1, val=ZERO, isSingular=1
  <65>   <37, 39>: opnd=xmm1, val=ZERO, isSingular=1
  <75>   <-1, 45>: opnd=qword ptr [r9+rax*8], val=ZERO, isSingular=1
  <83>   <50, 51>: opnd=xmm2, val=ZERO, isSingular=1
  <85>   <51, 52>: opnd=xmm2, val=ZERO, isSingular=1
  <88>   <52, 53>: opnd=xmm2, val=ZERO, isSingular=1
  <101>  <59, 60>: opnd=xmm1, val=ZERO, isSingular=1
==> detailed node info:
  [0]   vmovsd xmm0, qword ptr [r15+rax*8] @__upml_mod_MOD_upml_updateh._omp_fn.0[/public/home/csjt0800/lkl/.l
uild/build_base_mytest-m64.0000/UPML.fppized.f90:1490] [D][P]
  [1]   mov    rsi, qword ptr [rsp+0x10] @__upml_mod_MOD_upml_updateh._omp_fn.0[/public/home/csjt0800/lkl/.loca
ld/build_base_mytest-m64.0000/UPML.fppized.f90:1494]
  [2]   vmovsd xmm1, qword ptr [rsi+rax*8] @__upml_mod_MOD_upml_updateh._omp_fn.0[/public/home/csjt0800/lkl/.l
uild/build_base_mytest-m64.0000/UPML.fppized.f90:1494] [D][B]
  [3]   vsubsd xmm1, xmm1, qword ptr [rdx+0x08] @__upml_mod_MOD_upml_updateh._omp_fn.0[/public/home/csjt0800/l
d_s/build/build_base_mytest-m64.0000/UPML.fppized.f90:1494] [D][B]
  [4]   vmovsd xmm2, qword ptr [rcx+0x08] @__upml_mod_MOD_upml_updateh._omp_fn.0[/public/home/csjt0800/lkl/.lo
ild/build_base_mytest-m64.0000/UPML.fppized.f90:1494]
  [5]   mov    rdi, qword ptr [rsp+0x30] @__upml_mod_MOD_upml_updateh._omp_fn.0[/public/home/csjt0800/lkl/.loca
ld/build_base_mytest-m64.0000/UPML.fppized.f90:1494]
  [6]   vsubsd xmm2, xmm2, qword ptr [rdi+rax*8] @__upml_mod_MOD_upml_updateh._omp_fn.0[/public/home/csjt0800/
3d_s/build/build_base_mytest-m64.0000/UPML.fppized.f90:1494]
  [7]   vmulsd xmm2, xmm2, <rel> qword ptr [0x00000000006673e8] @__upml_mod_MOD_upml_updateh._omp_fn.0[/public
CPU/649.fotonik3d_s/build/build_base_mytest-m64.0000/UPML.fppized.f90:1494] [D][H]
  [8]   vfmadd132sd xmm1, xmm2, <rel> qword ptr [0x00000000006673d8] @__upml_mod_MOD_upml_updateh._omp_fn.0[/p
spec/CPU/649.fotonik3d_s/build/build_base_mytest-m64.0000/UPML.fppized.f90:1494] [D][H]
  [9]   vmulsd xmm1, xmm1, qword ptr [r12] @__upml_mod_MOD_upml_updateh._omp_fn.0[/public/home/csjt0800/lkl/.l
uild/build_base_mytest-m64.0000/UPML.fppized.f90:1494] [D][H]
  [10]  mov    rsi, qword ptr [rsp] @__upml_mod_MOD_upml_updateh._omp_fn.0[/public/home/csjt0800/lkl/.local/cp
ild_base_mytest-m64.0000/UPML.fppized.f90:1494]
  [11]  vfmadd231sd xmm1, xmm0, qword ptr [rsi] @__upml_mod_MOD_upml_updateh._omp_fn.0[/public/home/csjt0800/l
d_s/build/build_base_mytest-m64.0000/UPML.fppized.f90:1494]
```

```fortran
!$OMP DO PRIVATE(i,j,k)
do k=zstart,0
  do j=1,ny
    do i=1,nx

      Bxold = Bx_klow(i,j,k)

      Bx_klow(i,j,k) = ayh(j) * Bx_klow(i,j,k) +                          &
                       byh(j) * ((Ey(i,j,k+1)-Ey(i,j,k  ))*dzinv +        &
                                 (Ez(i,j,k  )-Ez(i,j+1,k))*dyinv)

      Hx(i,j,k) = azh(k) * Hx(i,j,k) +                                    &
                  bzh(k) * (cxe(i)*Bx_klow(i,j,k) - fxe(i)*Bxold) * muinv
      !------
      Byold = By_klow(i,j,k)

      By_klow(i,j,k) = azh(k) * By_klow(i,j,k) +                          &
                       bzh(k) * ((Ez(i+1,j,k)-Ez(i,j,k  ))*dxinv +        &
                                 (Ex(i,j,k  )-Ex(i,j,k+1))*dzinv)

      Hy(i,j,k) = axh(i) * Hy(i,j,k) +                                    &
                  bxh(i) * (cye(j)*By_klow(i,j,k) - fye(j)*Byold) * muinv
      !------
      Bzold = Bz_klow(i,j,k)

      Bz_klow(i,j,k) = axh(i) * Bz_klow(i,j,k) +                          &
                       bxh(i) * ((Ex(i,j+1,k)-Ex(i,j,k  ))*dyinv +        &
                                 (Ey(i,j,k  )-Ey(i+1,j,k))*dxinv)

      Hz(i,j,k) = ayh(j) * Hz(i,j,k) +                                    &
                  byh(j) * (cze(k)*Bz_klow(i,j,k) - fze(k)*Bzold) * muinv
    end do
  end do
end do
```

Thanks! Q&A