

# Tutorial: Exploring Software Inefficiency with Redundant Zeros



**Kelun Lei**

**Beihang University**

**Hands-on Tutorial @ CLUSTER24**



北京航空航天大学  
BEIHANG UNIVERSITY

# Outline

---

- Introduction
- Deep Analysis of Redundant Zeros
  - Pervasive Existence of Redundant Zeros
  - Root Causes of Redundant Zeros
- Detection of Redundant Zeros
- Evaluation
  - Experiment Setup
  - Overhead
- Hands-on Tutorial

# Outline

---

- Introduction
- Deep Analysis of Redundant Zeros
  - Pervasive Existence of Redundant Zeros
  - Root Causes of Redundant Zeros
- Detection of Redundant Zeros
- Evaluation
  - Experiment Setup
  - Overhead
- Hands-on Tutorial

# Redundant Zero - Example

```
1  for(int i=0; i<1000; ++i) {  
2    A[i] = 0; B[i] = i;  
3  }  
4  ...  
5  for(int i=0; i<1000; ++i) {  
6    C[i] = A[i] - B[i];  
7  }
```

Inefficient Codes

Memory operations  
frequently read zeros  
from the array A

Significant bytes of array  
B's values are always 0

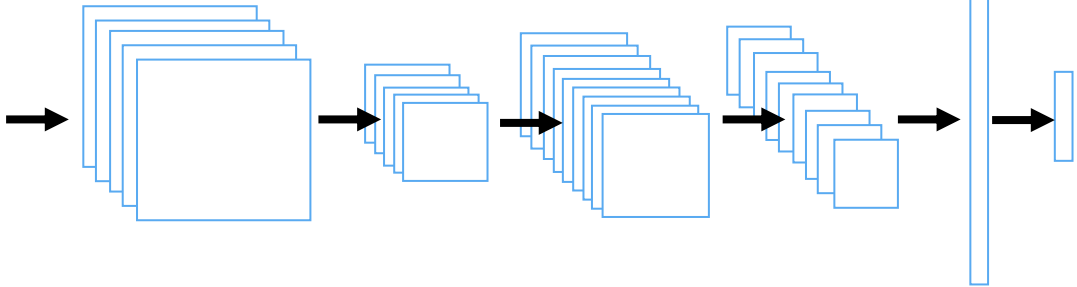
Observation

sparse data structure  
Avoid redundant loads

32-bit => 16-bit integer  
better cache usage &  
vectorization potentials

Optimization

Sparsity in Deep neural network



Hardware [A. Delmas Lascorz et.al., ASPLOS'19]  
Software [K. Peng et.al., SAMOS 2017]

A larger number of real-world applications have already been reported to contain a significant amount of redundant zeros and achieved significant speedup from the corresponding optimization.



Zero detection in HEVC  
[B. Lee et.al., IEEE Transactions on  
Multimedia, vol. 18, no.7, 2016]

# Outline

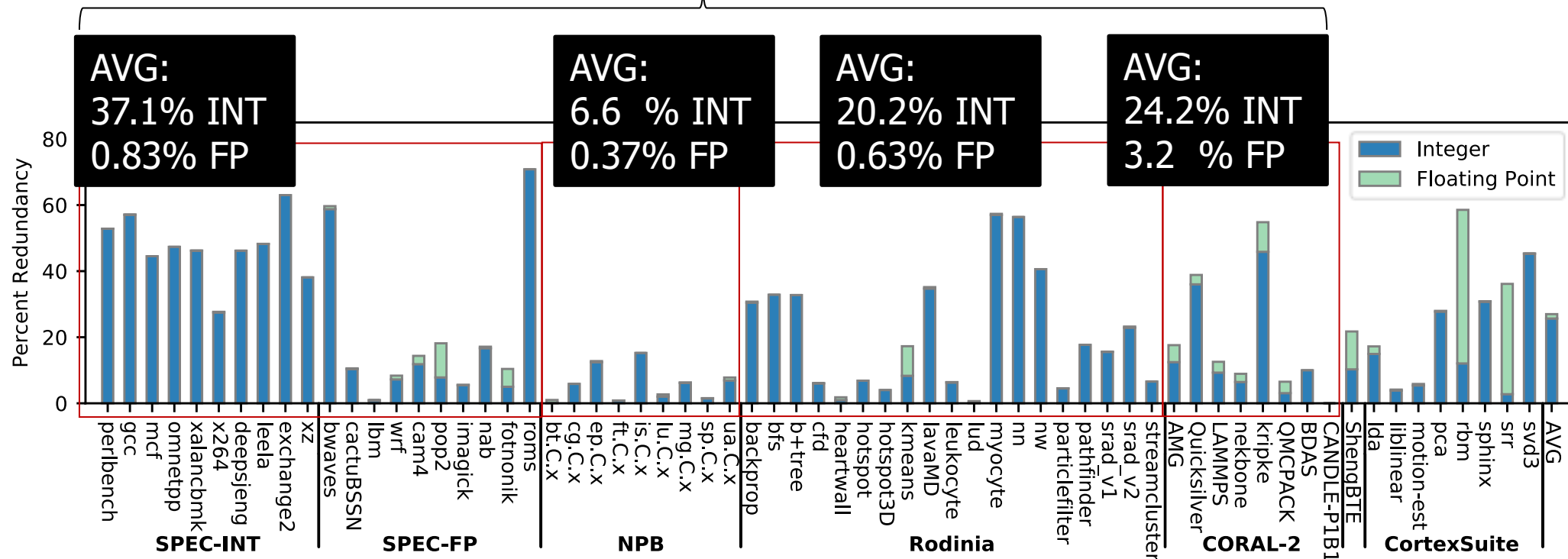
---

- Introduction
- Deep Analysis of Redundant Zeros
  - Pervasive Existence of Redundant Zeros
  - Root Causes of Redundant Zeros
- Detection of Redundant Zeros
- Evaluation
  - Experiment Setup
  - Overhead
- Hands-on Tutorial

# Pervasive Existence of Redundant Zeros

- Measure redundant zeros in SPEC CPU2017, NPB, Rodinia, and CORAL-2

Such large amounts of redundant zeros open an opportunity for code optimization **to avoid most redundant computations based on zeros.**

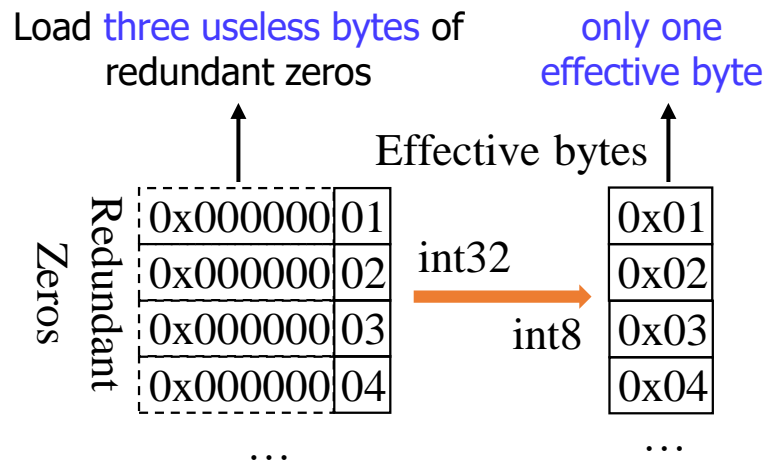


All these benchmarks are compiled with **gcc 9.2.0 -O3**

# Root Causes of Redundant Zeros

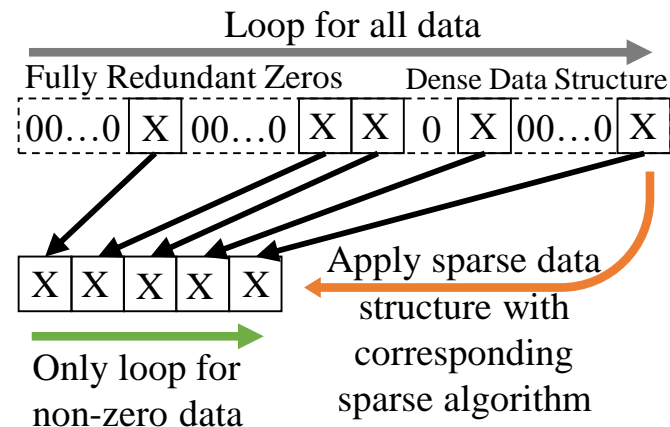
## Data with More than Enough Storage

- Redundant zeros **waste the limited resources** along the cache hierarchy
- Observed in *648.exchange2* (SPEC CPU2017)



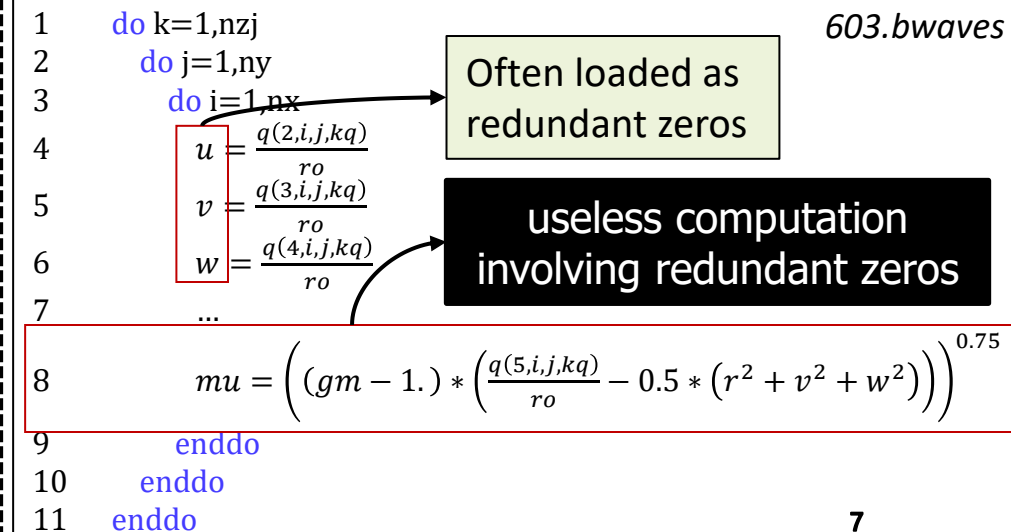
## Inappropriate Use of Data Structures

- A large fraction of fully redundant zeros with the data structures can be detected during the execution.
- Sparse data using dense data structure results in **useless loads and computation instructions**
- Observed in *649.fotnonik* (SPEC CPU2017)



## Zero-agonistic Computation

- instruction level**: instructions load redundant zeros from memory and compute with these redundant zeros.
- The useless computation with redundant zeros worth performance optimization, especially **when the code region involves heavy computation**.
- Observed in *644.nab* (SPEC CPU2017), *heartwall* (Rodinia), *ShengBTE*



# Outline

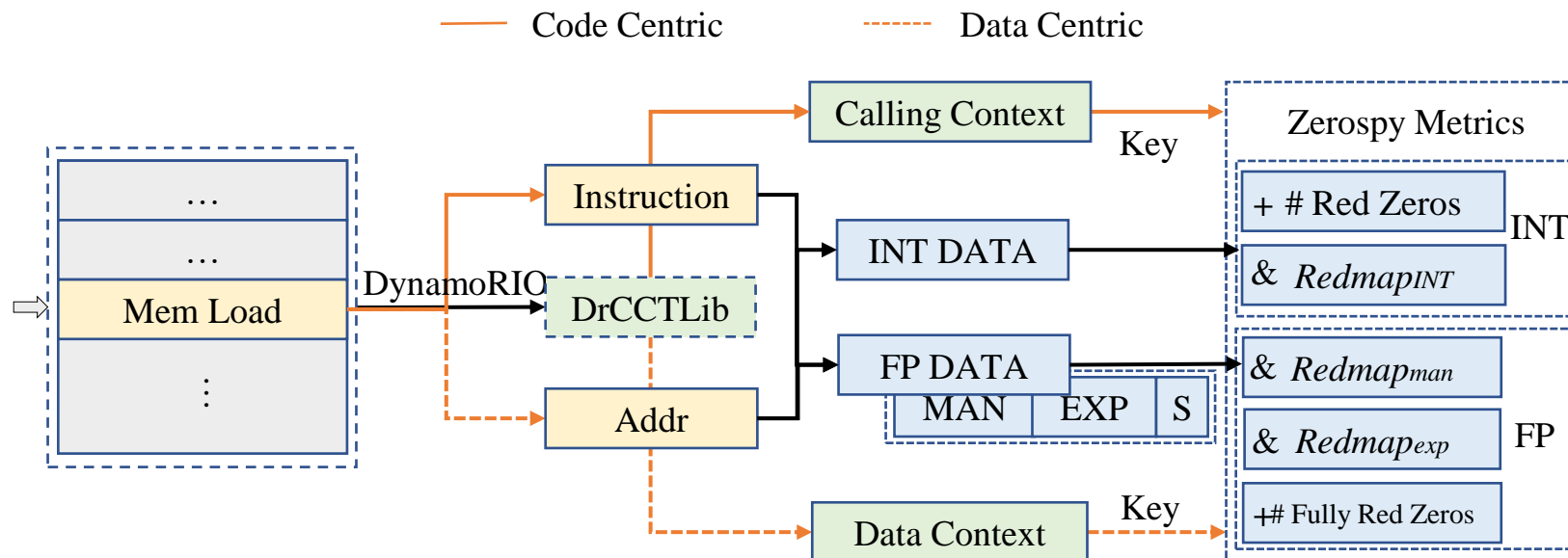
---

- Introduction
- Deep Analysis of Redundant Zeros
  - Pervasive Existence of Redundant Zeros
  - Root Causes of Redundant Zeros
- Detection of Redundant Zeros
- Evaluation
  - Experiment Setup
  - Overhead
- Hands-on Tutorial

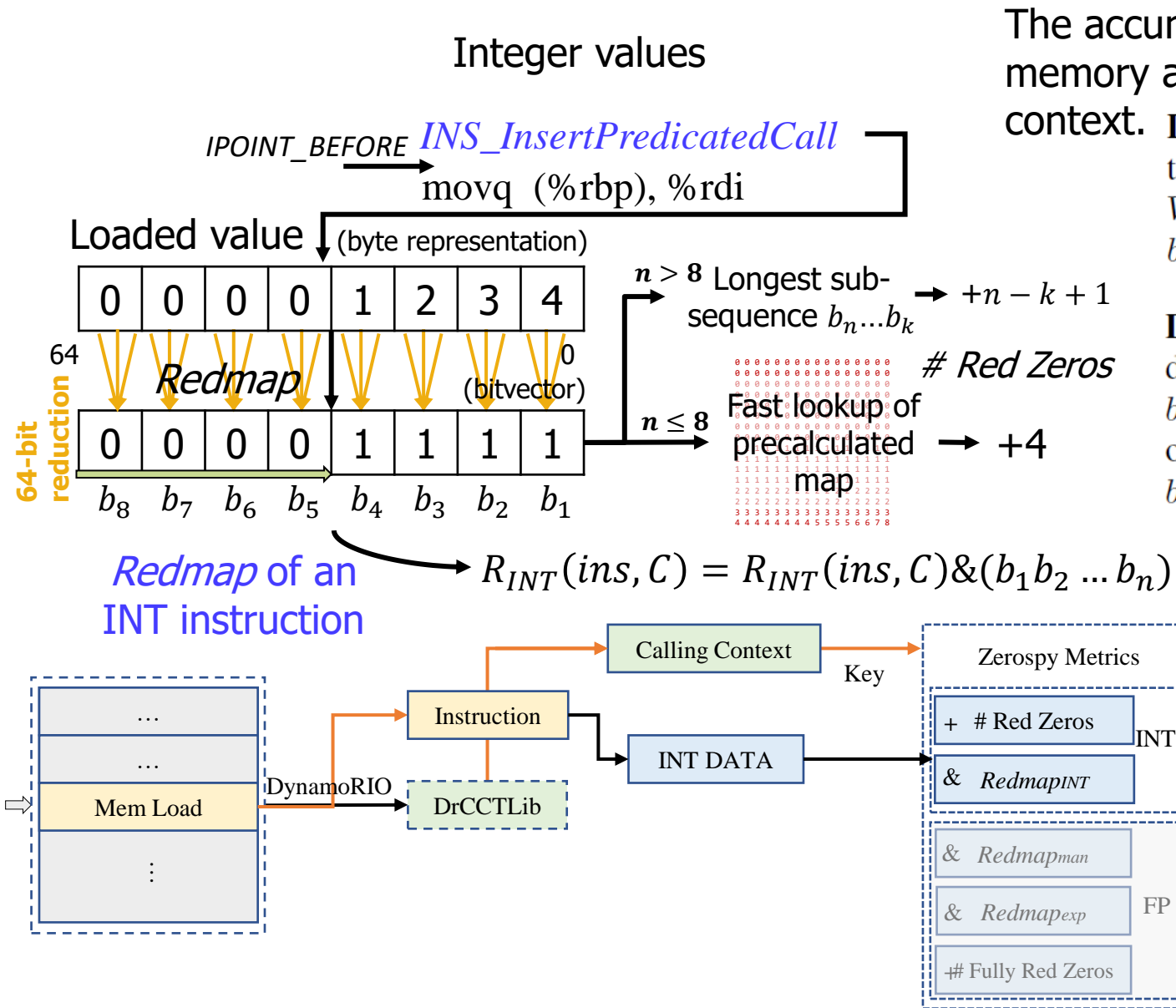


# Detection of Redundant Zeros

- Manually identifying redundant zeros used in programs is tedious, which requires strong domain knowledges
- We develop a tool, ZeroSpy, to automate the analysis for fully optimized binary code
  - Code-centric : Instruction
  - Data-centric : Data objects
  - Focus on detection and optimization of redundant zeros in memory



# Code-centric Analysis - Instruction



The accumulated redmap of an instruction indicates the memory access pattern of the instruction in its calling context.

**Definition 1 (Redmap).** Let  $B_1 B_2 \dots B_n$  be the byte representation of a value  $V$ , where  $B_n$  is the most significant byte of  $V$ . The *redmap* of  $V$  is defined as a **bit vector**  $b_1 b_2 \dots b_n$ , where  $b_i$  ( $i = 1, 2, \dots, n$ ) is bit 0 if  $B_i = 0$ , otherwise  $b_i$  is bit 1.

**Definition 2 (Redundant Zero).** A value  $V$  contains redundant zero when a sub-sequence  $b_k \dots b_n$  exists in the *redmap*  $b_1 b_2 \dots b_n$  of  $V$ , where  $b_i = 0$  ( $1 \leq k \leq i \leq n$ ). The number of redundant zeros in  $V$  is defined as the length of the longest  $b_k \dots b_n$ . If the value  $V = 0$ , we call it *fully redundant zeros*.

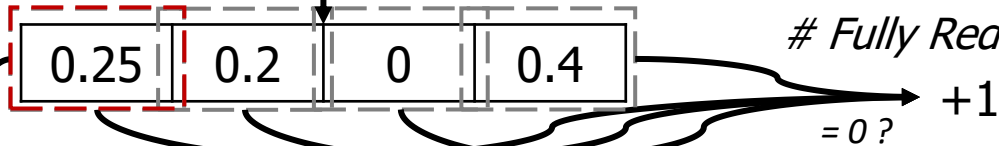
- Code-centric : Instruction loads integer value
- Insert INT routine to analyze
- Obtain redmap according to Definition 1
- Count the redundant zeros byte to byte according to Definition 2
- Accumulate the redmap of instruction with the same calling context

# Code-centric Analysis - Instruction

Floating-point values

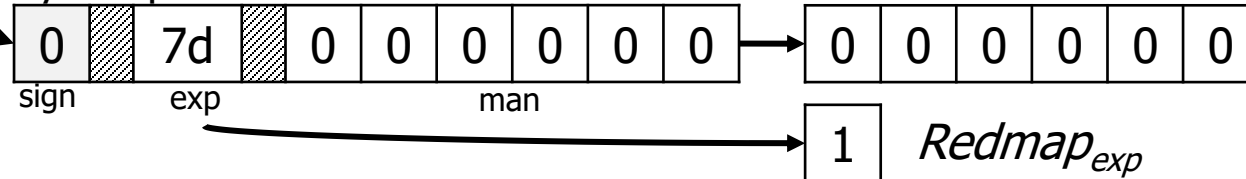
*IPOINT\_BEFORE* *INS\_InsertPredicatedCall*  
`movapsx -0x50(%rsi), %xmm5`

Loaded value



The number of redundant zeros of a single floating-point value has little interest for performance optimization because we cannot downgrade its representation (using fewer bits) without loss of accuracy.

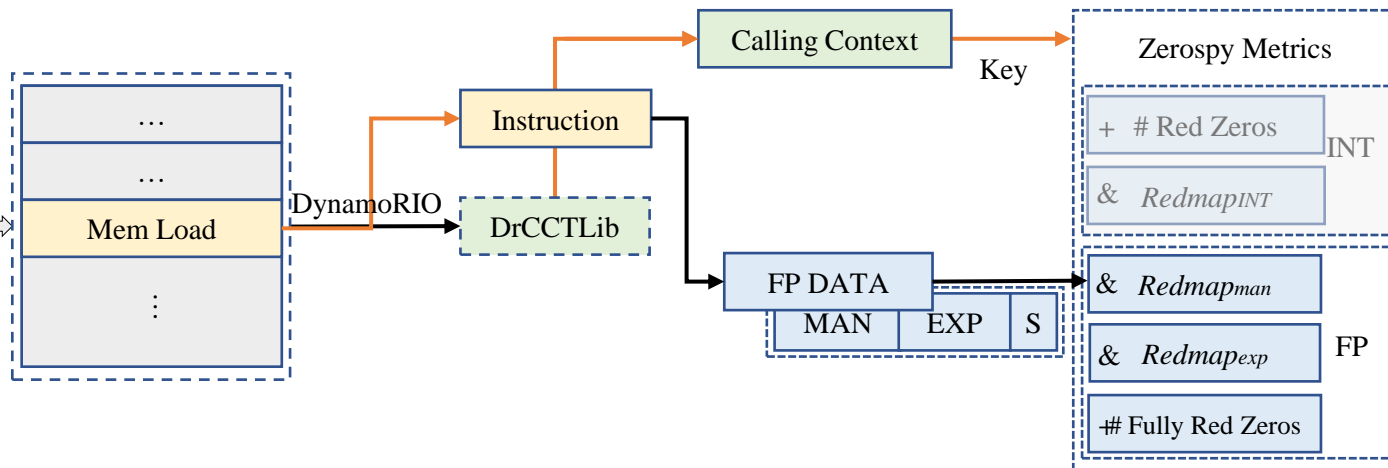
Byte representation IEEE floating-point format



$$R_{FP}(ins, C) = \begin{cases} R_{man}(ins, C) = R_{man}(ins, C) \& (b_1 b_2 \dots b_k) \\ R_{exp}(ins, C) = R_{exp}(ins, C) \& (b_{k+1} \dots b_n) \end{cases}$$

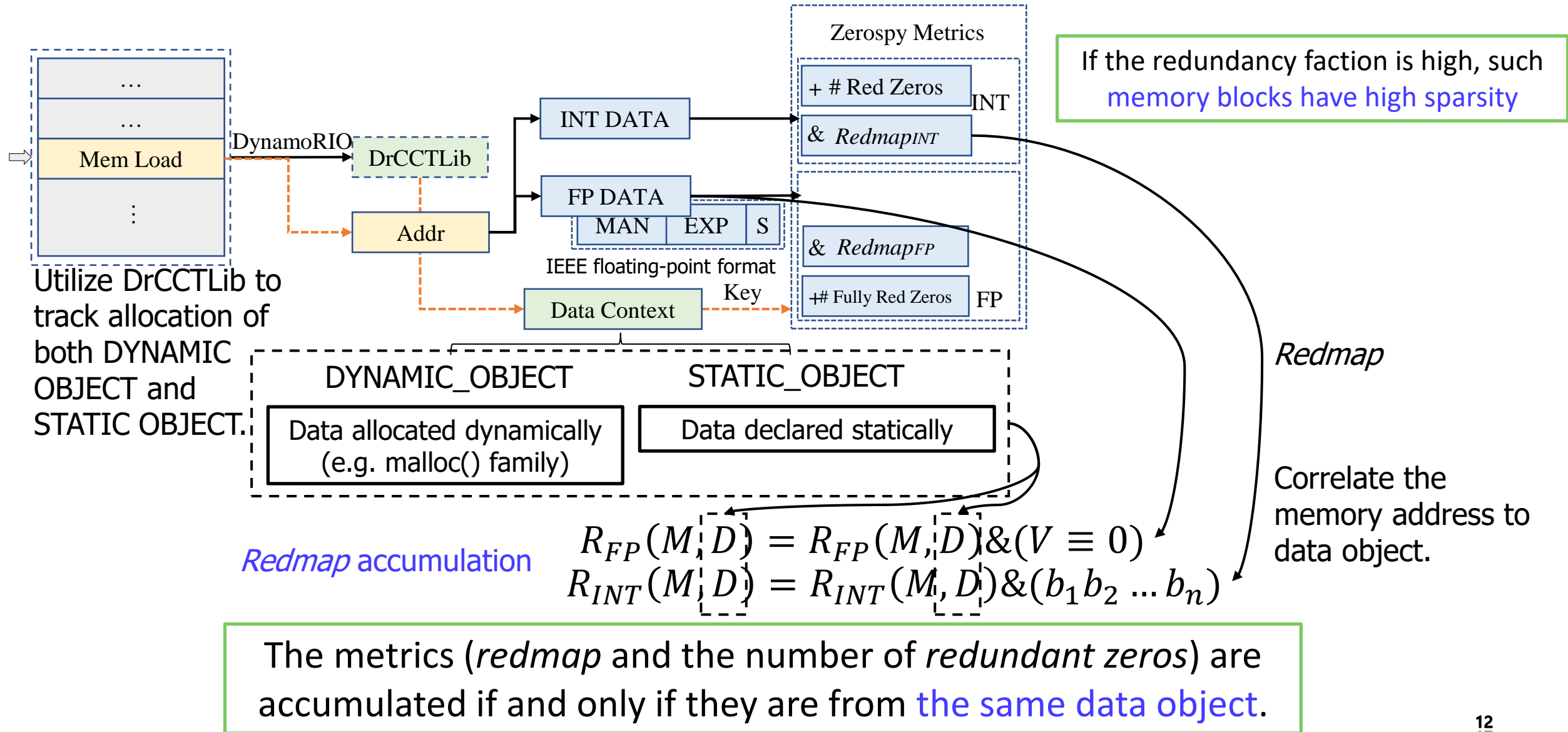
*Redmap<sub>man</sub>* *Redmap of an FP instruction*

$k = 23; n = 31$  for single  
 $k = 52; n = 63$  for double



- Code-centric : Instruction loads floating-point value
- Insert FLOAT routine to analyze
- Only mantissa and exponent are the potential targets for software inefficiency when they contain redundant zeros.

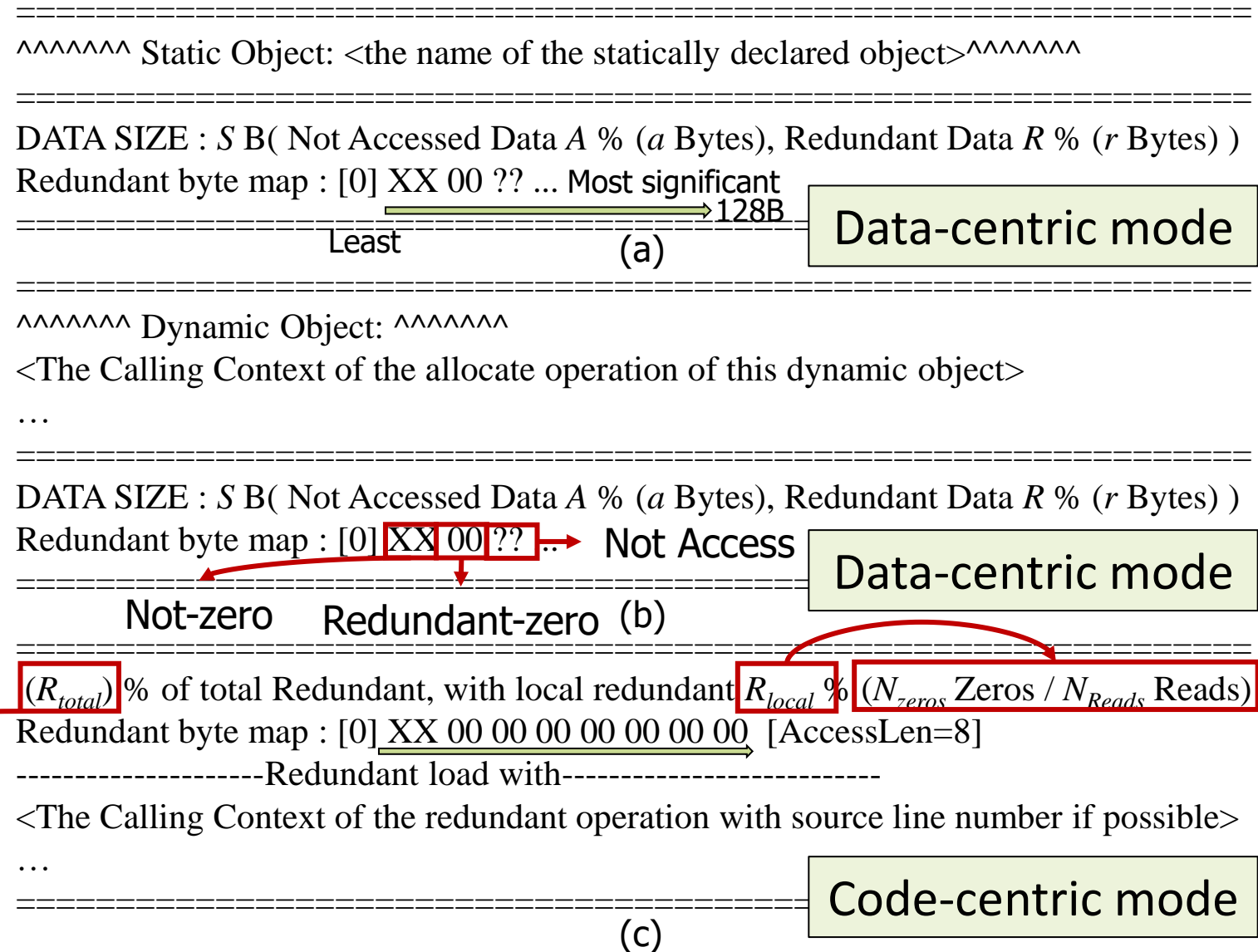
# Data-centric Analysis - Data Object Sparsity



# Reporting Redundant Zeros

- Reported redundant zeros by ZeroSpy
  - Present the top few candidates
    - redundancy fraction
    - redmap
  - Static object
  - Dynamic object
  - Integer/floating-point instructions

$$\frac{\text{\# redundant zeros of the reported instruction}}{\text{\# all detected redundant zeros}}$$



# Outline

---

- Introduction
- Deep Analysis of Redundant Zeros
  - Pervasive Existence of Redundant Zeros
  - Root Causes of Redundant Zeros
- Detection of Redundant Zeros
- Evaluation
  - Experiment Setup
  - Overhead
- Hands-on Tutorial

# Evaluation

- SPEC CPU2017, CORAL-2, NAS Parallel Benchmarks (NPB), Rodinia, CortexSuite, software package ShengBTE.
  - SPEC CPU2017
  - CORAL-2, NPB, CoretexSuite, Rodinia
  - ShengBTE

Machine	Intel-Broadwell
Processor	Xeon E5-2680v4@2.4GHz
Cores	14
Memory	256 GB DDR4
L1/L2/L3 Cache	448KB/3.5MB/35MB
Compiler	gcc 9.2.0 -O3
System	Linux 3.10.0-514.el7.x86_64

Redundancy types	Programs	Directed by	Problematic procedures/variables	speedup	power saving
Data with more than enough storage	641.leela [42]	CC	FastState.cpp:176	8.7%	7.57%
	648.exchange2 [42]	CC	brute_force_mp_block	9.8%	5.90%
	JM [55]	CC/DC	bi_context_type	13.0%	14.1%
Inappropriate use of data structure	649.fotonik3d [42]	DC/CP	E_z:yeemain.fppized.f90:112	52.8%	34.85%
	gcc (SPEC CPU2006) [42]	DC	last_set:loop_regs_scan()	11.3%	9.8%
	srr [50]	CC	SRREngine.c:25	58.8%	38.7%
	BT [49]	DC	work_lhs:z_solver.f	4.4%	4.5%
Useless computation	644.nab [42]	CC	eff.c:189	7.1%	5.88%
	heartwall [44]	CC	kernel.c:57	9.6%	6.79%
	ShengBTE [51]	CC	ShengBTE.f90:476	9.46%	10.04%
	638.imagick [42]	CC	morphology.c:2982	25%	28.8%
	NWChem [56]	DC/CP	tce mo2e trans.F:240	20%	10%
	backprop [44]	CC	backprop.c:323	40.8%	29.4%
	Stack RNN [57]	CC	StackRNN.h:loop(352,365)	26.80%	26.73%
	xgboost [58]	CC	updater_colmaker.cc:if(422,434)	3.03%	2.55%
	QuEST [59]	CC	QuEST_cpu.c:2120	8.77%	8.93%

\* Code-Centric Mode (CC), Data-Centric Mode (DC), Cacheline level and Page level detection (CP)



# Overhead

Program	% Redundancy		Time Overhead		Memory Overhead	
	INT	FP	CC	DC	CC	DC
600.perlbench	53.91	0.02	72.70	148.02	4.78	3.59
	51.10	0.03	87.80	227.26	4.53	3.00
	53.45	0.00	75.19	224.89	9.29	6.80
602.gcc	52.88	0.04	66.44	180.22	20.40	12.07
	59.51	0.18	86.66	156.20	53.63	33.66
	58.73	0.15	85.63	157.99	54.89	30.79
605.mcf	44.51	0.00	28.31	55.54	1.05	1.71
620.omnetpp	47.36	0.00	29.96	120.39	2.83	3.22
623.xalancbmk	46.19	0.03	38.19	174.72	1.72	3.43
625.x264	27.67	0.06	57.71	146.08	2.97	1.76
	27.08	0.34	60.09	149.78	9.22	4.28
	27.66	0.25	62.34	162.06	3.95	2.44
631.deepsjeng	46.19	0.00	335.05	232.79	1.03	1.26
641.leela	48.21	0.05	45.91	38.92	225.25	151.70
648.exchange2	63.00	0.00	80.52	81.31	169.23	42.25
657.xz	34.97	0.00	32.19	57.35	1.14	3.17
	41.26	0.00	26.81	46.88	1.26	1.88
SPEC INT MAX	63.00	0.34	335.05	232.79	225.25	151.70
603.bwaves	58.35	0.92	100.98	93.11	1.21	1.70
	59.17	0.90	88.16	94.91	1.21	1.71
607.cactuBSSN	10.24	0.24	60.17	94.80	1.42	1.10
619.lbm	0.93	0.03	12.73	18.54	1.70	4.95
621.wrf	7.23	1.14	15.14	29.48	22.34	11.06
627.cam4	11.84	2.52	27.02	68.24	5.82	4.58
628.pop2	7.83	10.33	44.79	85.14	2.15	2.50
638.imagick	5.45	0.17	86.06	216.73	1.83	8.49
644.nab	16.67	0.46	75.91	171.00	5.18	7.39
649.fotnonik	5.01	5.39	23.98	59.93	1.24	2.92
654.roms	70.82	0.31	25.07	67.65	1.16	2.72
SPEC FP MAX	70.82	10.33	100.98	216.73	22.34	11.06

NPB MEDIAN	5.88	0.18	59.17	78.72	3.97	15.02
NPB MAX	15.25	0.89	196.09	229.35	148.30	29.77
Rodinia MEDIAN	15.57	0.03	66.84	84.79	62.86	15.73
Rodinia MAX	57.00	8.95	450.39	524.62	2656.65	48.70
AMG	11.39	5.05	25.97	66.29	5.02	4.40
	13.52	5.22	34.92	71.58	27.13	3.55
Quicksilver	36.01	2.84	64.55	159.55	1.62	2.99
LAMMPS	9.29	3.27	39.43	76.43	2.50	19.47
nekbone	6.45	2.47	79.55	295.02	2.53	45.08

- AVG:
  - Code centric: time 72.31x, memory 108.94x
  - Data centric: time 118.51x, memory 14.26x
- the profiling overhead appears to be high when there is a large fraction of redundant zeros existing in the program
  - a dynamic profiler based on instrumentation
  - the runtime overhead is similar to the well-accepted profilers



# Outline

---

- Introduction & Related Works
- Deep Analysis of Redundant Zeros
- Detection of Redundant Zeros
- Evaluation
- Hands-on Tutorial
  - Installation
  - Case Study – backprop
  - Case Study – EP
  - Case Study – 648.exchange2
  - Case Study – 644.nab

# Installation

- Install with source code:

- Dependencies:

git, gcc>=9, g++, make, cmake>=3.20

- Source Code:

/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:

- ZeroSpy:

\$DRRUN -t zerospy -- <EXE> <ARGS>

# Case Study – backprop Benchmark (~7mins)

- Get the Benchmark and Compile: backprop - machine learning/backward propagation

- Get the Rodinia Benchmark:

```
cp /public/home/buaa_hipo/shared_folder/\
backprop ./
```

- Compile:

```
cd backprop && make
```

- Original run and profiling run:

- Fill in the desired commands into the template

- Job template:

```
NTASKS=1
#!/bin/bash
JOBNAME="zerospy-backprop"
DRRUN="`pwd`/Vclinic/build/bin64/drrun"
# tool name example: zerospy, trivialspy
TOOL="zerospy"
# TARGET command for profiling: CMD="<EXE> <ARGS>"
CMD="`pwd`/backprop/backprop 6553600"
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
# profiling run
time $DRRUN -t $TOOL -- $CMD

END
```

# Case Study – backprop Benchmark

- Unoptimized execution timing:

```
time ./backprop 6553600
```

- Profiling the backprop with ZeroSpy:

```
$DRRUN -t zerospy -- ./backprop 65536
```

- Resulting files are generated in the x86-<host>-<PID>-zerospy folder
  - zerospy.log is the summary of redundant zero metrics
  - thread-<id>.log is the detailed per-thread reports.
  - Summary reports: backprop has more than 10% integer and floating-point redundant zeros.

## **Original** runtime:

```
Random number generator seed: 7
Input layer size : 6553600
Starting training kernel
Performing CPU computation
Training done
real    0m4.590s
user    0m4.057s
sys     0m0.526s
```

### #THREAD 1 Redundant Read:

```
TotalBytesLoad: 633528008
RedundantBytesLoad: 82279887 12.99
ApproxRedundantBytesLoad: 104852368 16.55
...
```

### #THREAD 2 Redundant Read:

```
TotalBytesLoad: 634117696
RedundantBytesLoad: 82711454 13.04
ApproxRedundantBytesLoad: 104852368 16.54
```

[zerospy.log](#)

# Case Study – backprop Benchmark

- Optimization guidance: significant fully redundant zeros in backprop.c:323 (exp and man are 0).
- After further analysis, we found that delta and oldw are often 0

----- Dumping Approximation Redundancy Info -----

Total redundant bytes = 16.550550 %

INFO : Total redundant bytes = 16.550550 % (104852368 / 633528008)

=====**(12.499998) % of total Redundant, with local redundant 100.000000 %**  
**(13106544 Zeros / 13106544 Reads)**=====

=====**Redundant byte map : [ sign | exponent | mantissa ]**=====

XX | **00** | **00 00 00**

=====**[AccessLen=4, typesize=4]**=====

-----Redundant load with-----

#0 0x0000000000400d60 "cvtss2sd xmm0, dword ptr [rdi]" in  
bpnn\_adjust\_weights\_omp\_fn.0 at  
[/public/home/csjt0800/lkl/.local/rodinia\_3.1/openmp/backprop/**backprop.c:323**]

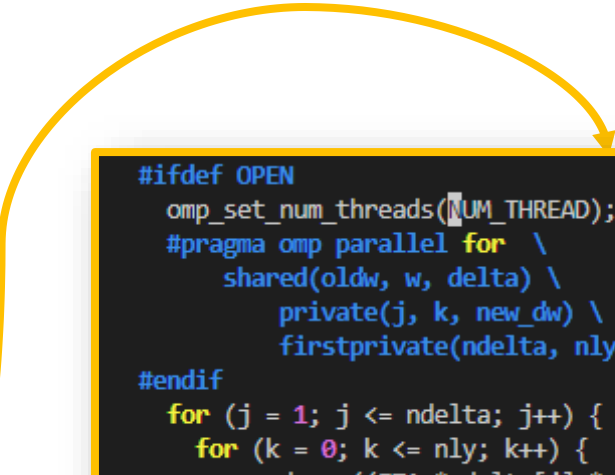
#1 0x00002b3889e0e403 "call r12" in gomp\_thread\_start at [:0]

#2 0x00002b3889e6ddcd "call qword ptr [fs:0x00000640]" in start\_thread at  
[/usr/src/debug/glibc-2.17-c758a686/nptl/pthread\_create.c:307]

#3 0x00002b388a19deab "call rax" in \_\_clone at  
[./sysdeps/unix/sysv/linux/x86\_64/clone.S:111]

#4 0xfffffffffffffffe "<NULL>" in THREAD[1]\_ROOT\_CTX at [<NULL>:0]

#5 0x0000000000000000 "<NULL>" in PROCESS[105469]\_ROOT\_CTX at  
[<NULL>:0]



```
#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;
    }
}
```

# Case Study – backprop Benchmark

- Optimization guidance: significant fully redundant zeros in backprop.c:323 (exp and man are 0).
- After further analysis, we found that delta and oldw are often 0

```
#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;
    }
}
```

optimize



```
#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++) {
    if(delta[j]==0) {
        for (k = 0; k <= nly; k++) {
            if(oldw[k][j]!=0) {
                new_dw = (MOMENTUM * oldw[k][j]);
                w[k][j] += new_dw;
                oldw[k][j] = new_dw;
            }
        }
    } else {
        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;
        }
    }
}
```

# Case Study – backprop Benchmark

Modify the corresponding file and re-compile the backprop benchmark to check the optimization effects.

```
Random number generator seed: 7
Input layer size : 6553600
Starting training kernel
Performing CPU computation
Training done
```

```
real    0m4.590s
user    0m4.057s
sys     0m0.526s
```

**Speedup 1.35x**



```
Random number generator seed: 7
Input layer size : 6553600
Starting training kernel
Performing CPU computation
Training done
```

```
real    0m3.409s
user    0m2.923s
sys     0m0.457s
```

# Case Study – EP Benchmark (~10mins)

- Get the Benchmark and Compile: EP - Embarrassingly Parallel Problem

- Get the NPB Benchmark:

```
cp /public/home/buaa_hipo/shared_folder/\
NPB3.4.2 / ./
```

- Compile:

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

```
cd NPB3.4.2/NPB3.4-OMP/  
cp config/make.def.template config/make.def  
vim config/make.def # add the -g to all flags  
make EP CLASS=C
```

- Original run and profiling run:
  - Fill in the desired commands into the template

- Job template:

```
NTASKS=1  
#!/bin/bash  
JOBNAME="zerospy-ep"  
DRRUN="`pwd`/Vclinic/build/bin64/drrun"  
# tool name example: zerospy, trivialspy  
TOOL="zerospy"  
# TARGET command for profiling: CMD="<EXE> <ARGS>"  
CMD="`pwd`/NPB3.4.2/NPB3.4-OMP/bin/ep.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  
# profiling run  
time $DRRUN -t $TOOL -- $CMD  
  
END
```



# Case Study – EP Benchmark

- Unoptimized execution timing:

```
time ./bin/ep.C.x
```

- Profiling the ep with ZeroSpy:

```
$DRRUN -t zerospy -- ./bin/ep.C.x
```

- Resulting files are generated in the x86-<host>-<PID>-zerospy folder
  - zerospy.log is the summary of redundant zero metrics
  - thread-<id>.log is the detailed per-thread reports.
  - Summary reports: backprop has more than 20% floating-point redundant zeros.

EP Benchmark Completed.

Class	=	C
Total threads	=	32
Avail threads	=	32
Verification	=	SUCCESSFUL
Version	=	3.4.2
Compile date	=	19 Sep 2024

Compile options:

FC	=	gfortran
FLINK	=	\$(FC)
F_LIB	=	(none)
F_INC	=	(none)
FFLAGS	=	-O3 -fopenmp -g
FLINKFLAGS	=	\$(FFLAGS)
RAND	=	randi8
real		0m12.777s
user		5m47.013s
sys		0m0.457s

## #THREAD 1 Redundant Read:

TotalBytesLoad: 47449361400

RedundantBytesLoad: 3061575375 6.45

ApproxRedundantBytesLoad: 9594559840 20.22

...

## #THREAD 2 Redundant Read:

TotalBytesLoad: 47452444416

RedundantBytesLoad: 3062222002 6.45

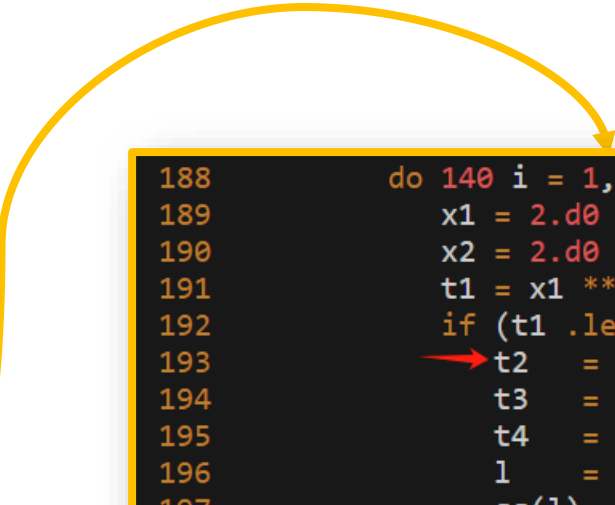
ApproxRedundantBytesLoad: 9595085744 20.22

[zerospy.log](#)

# Case Study – EP Benchmark

- Optimization guidance: significant fully redundant zeros in ep.f90:193.
- After further analysis, we found that t1 often equals to 1 which makes t2 often equals to 0.

```
***** Dump Data from Thread 0 *****
===== (27.452865) % of total Redundant, with local redundant 100.000000 %
(843363280 Bytes / 843363280 Bytes) =====
===== with All Zero Redundant 100.000000 % (105420410 / 105420410)
=====
===== Redundant byte map : [0] 00 00 00 00 00 00 00 00 [AccessLen=8]
=====
-----Redundant load with-----
#0 0x00002b17a01185df "pop rbp" in __ieee754_log_sse2 at
[./sysdeps/ieee754/dbl-64/e_log.c:214]
#1 0x0000000000401041 "call 0x0000000000400d60" in MAIN . omp fn.1 at
[/public/home/csjt0800/lkl/.local/vclinic_cases_arm/NPB3.4.2/NPB3.4-
OMP/org/EP/ep.f90:193]
```



```
188      do 140 i = 1, nk
189          x1 = 2.d0 * x(2*i-1) - 1.d0
190          x2 = 2.d0 * x(2*i) - 1.d0
191          t1 = x1 ** 2 + x2 ** 2
192          if (t1 .le. 1.d0) then
193              → t2 = sqrt(-2.d0 * log(t1) / t1)
194              t3 = abs(x1 * t2)
195              t4 = abs(x2 * t2)
196              l = max(t3, t4)
197              qq(1) = qq(1) + 1.d0
198              sx = sx + t3
199              sy = sy + t4
200          endif
201      140 continue
```

# Case Study – EP Benchmark

- Optimization guidance: significant fully redundant zeros in ep.f90:193.
- After further analysis, we found that t1 often equals to 1 which makes t2 often equals to 0.

```
188      do 140 i = 1, nk
189          x1 = 2.d0 * x(2*i-1) - 1.d0
190          x2 = 2.d0 * x(2*i) - 1.d0
191          t1 = x1 ** 2 + x2 ** 2
192          if (t1 .le. 1.d0) then
193              → t2 = sqrt(-2.d0 * log(t1) / t1)
194              t3 = abs(x1 * t2)
195              t4 = abs(x2 * t2)
196              l = max(t3, t4)
197              qq(1) = qq(1) + 1.d0
198              sx = sx + t3
199              sy = sy + t4
200          endif
201 140      continue
```

optimize

Speedup 1.10x

```
188      do 140 i = 1, nk
189          x1 = 2.d0 * x(2*i-1) - 1.d0
190          x2 = 2.d0 * x(2*i) - 1.d0
191          t1 = x1 ** 2 + x2 ** 2
192          if (t1 .le. 1.d0) then
193              if(t1 .eq. 1.d0) then
194                  t2 = 0
195                  t3 = 0
196                  t4 = 0
197                  l = 0
198                  qq(0) = qq(0) + 1.d0
199              else
200                  t2 = sqrt(-2.d0 * log(t1) / t1)
201                  t3 = abs(x1 * t2)
202                  t4 = abs(x2 * t2)
203                  l = max(t3, t4)
204                  qq(1) = qq(1) + 1.d0
205                  sx = sx + t3
206                  sy = sy + t4
207              endif
208          endif
209 140      continue
```

# Case Study – exchange2 (Optional)

- exchange2 - Sudoku Puzzle Generator

- Get the SPEC CPU2017 Benchmark:

```
cp /public/home/buaa_hipo/shared_folder/648.exchange2_s/ ./
```

- Compile and Profiling:

```
cd 648.exchange2_s/  
bash make.clean.out  
bash make.out  
# original run  
time ./run.sh  
# profiling run  
$DRRUN -t zerospy -- ./exchange2_s 0
```

```
! Declare block as one-byte integer:  
integer(kind=1) :: block(:, :, :)  
integer(kind=1) :: block(r, r, r)
```

```
10 subroutine new_solver(part, block, complete, key, changed)  
11 implicit none  
12 integer :: i, j, k, row, col, val, boxr, boxc, remember, block(:, :, :), part(:, :), &  
13     cycles, non_zeroes, difficulty_index, key, subblock  
14 logical :: complete, changed, expensive  
840 module brute_force  
841 ! (c) Michael Metcalf, 2006.  
842 ! Runs at about 1.5msecs per easy puzzle on a 1GHz CPU, 40msecs for medium/hard ones,  
843 ! and 400msecs for very hard ones.  
844 use logic  
845 implicit none  
846 private  
847 public brute, covered, soln, pearl  
848 integer, parameter :: r=9  
849 integer :: sudoku1(r, r), i, j, sudoku2(r, r), sudoku3(r, r), soln, block(r, r, r), &  
850 val, bc, br
```

From the pattern in the redmap, we can infer that the redundant zeros come from the data type that is more than enough to store the value, for **the most significant three bytes** of the integer data type are always zero.

----- Dumping INTEGER Redundancy Info -----

Total redundant bytes = 53.678792 %

INFO : Total redundant bytes = 53.678792 % (54963859103 / 102393993790)

===== (1.640774) % of total Redundant, with local redundant 75.000000 %  
(901832781 Bytes / 1202443708 Bytes) =====

===== with All Zero Redundant 0.000000 % (0 / 300610927) =====

===== **Redundant byte map : [0] XX 00 00 00 [AccessLen=4]** =====

-----Redundant load with-----

```
#0 0x0000000000041ab46 "cmp dword ptr [rsp+0x00000150], r13d" in  
_brute_force_MOD_digits_2 at  
[/public/home/csjt0800/lkl/.local/cpu2017/benchspec/CPU/648.exchange2_s/build/build_base_mytest-m64.0000/exchange2.fppized.f90:1079]  
#1 0x0000000000041afbf "call 0x00000000000419b9a" in  
_brute_force_MOD_digits_2 at  
[/public/home/csjt0800/lkl/.local/cpu2017/benchspec/CPU/648.exchange2_s/build/build_base_mytest-m64.0000/exchange2.fppized.f90:1129]
```

Reducing the data type to one-byte integer achieves 1.06x speedup

# Case Study – nab (Optional)

- Get the Benchmark and Compile: nab is a benchmark from SPEC CPU2017 which performs molecular modeling in life science simulation.
- We use ref input and the profiling process is time-consuming since this program has significant number of memory reads.

local redundant **100.000000** %  
==== Redundant byte map : [ sign | exponent | mantissa ] =====  
00 | 00 00 | 00 00 00 00 00 00 00 , XX | 00 00 | 00 00 00 00 00 00 00  
==== [AccessLen=16, typesize=8] =====  
-----Redundant load with-----  
#0 0x00000000000406e39 "vxorpd xmm2, xmm2, <rel> oword ptr  
[0x00000000000432b10]" in egb.\_omp\_fn.1 at  
[/public/home/csjt0800/lkl/.local/cpu2017/benchspec/CPU/644.nab\_s/build/buil  
d\_base\_mytest-m64.0000/**eff.c:2314**]

- Optimzation
  - Skip useless computation
  - 7.1% speedup
  - 5.88% power saving

The redundant zeros in the *egb* function of nab

```
1  for (j = 0; j < prm->Natom; j++) {  
2    for (k = 0; k < npairs; k++) {  
3      ...  
4      fgbk = - (*kappa) * KSCALE / fgbi;  
5      expmkf = exp(fgbk) / (*diel_ext);  
6      temp6 = qiqj * temp4 * (dielfac + fgbk * expkmf);  
7      ...  
8    }  
9  }
```

```
1  if (*kappa==0) {  
2    for (j = 0; j < prm->Natom; j++) {  
3      for (k = 0; k < npairs; k++) {  
4        ...  
5        expmkf = 1 / (*diel_ext);  
6        temp6 = qiqj * temp4 * dielfac;  
7        ...  
8      }  
9    }  
10 } else {  
11   ... // same as before  
12 }
```

---

**Thanks!**  
**kelunlei@buaa.edu.cn**