2017 IEEE International Symposium on Workload Characterization

LORE: A Loop Repository for the Evaluation of Compilers

Zhi Chen*, Zhangxiaowen Gong†, Justin Josef Szaday†, David C. Wong‡, David Padua†, Alexandru Nicolau*, Alexander V Veidenbaum*, Neftali Watkinson*, Zehra Sura§, Saeed Maleki¶, Josep Torrellas†, Gerald DeJong†

‡Intel Corporation, §IBM Research, ¶Microsoft Research,
*University of California, Irvine, †University of Illinois at Urbana-Champaign

Background

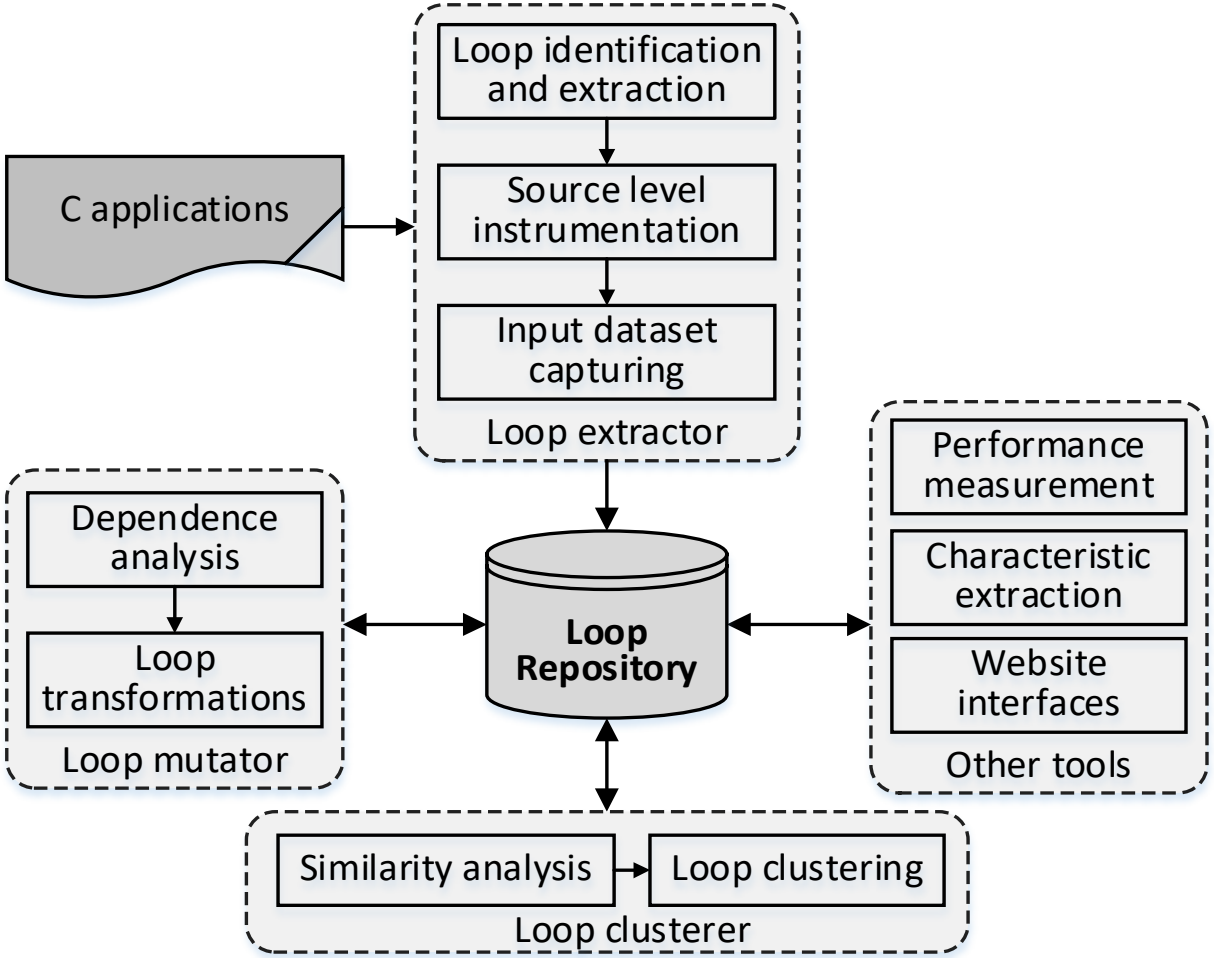
- Loop transformation
  - Better locality, less computation, and parallelism/vectorization
  - Advantage
    - Programmers can focus on preparation of the readable and correct algorithm
  - Challenges
    - Will compilers generate efficient code?
    - Is it necessary to manually transform code?
  - Much room for advances in compiler technology

Motivation

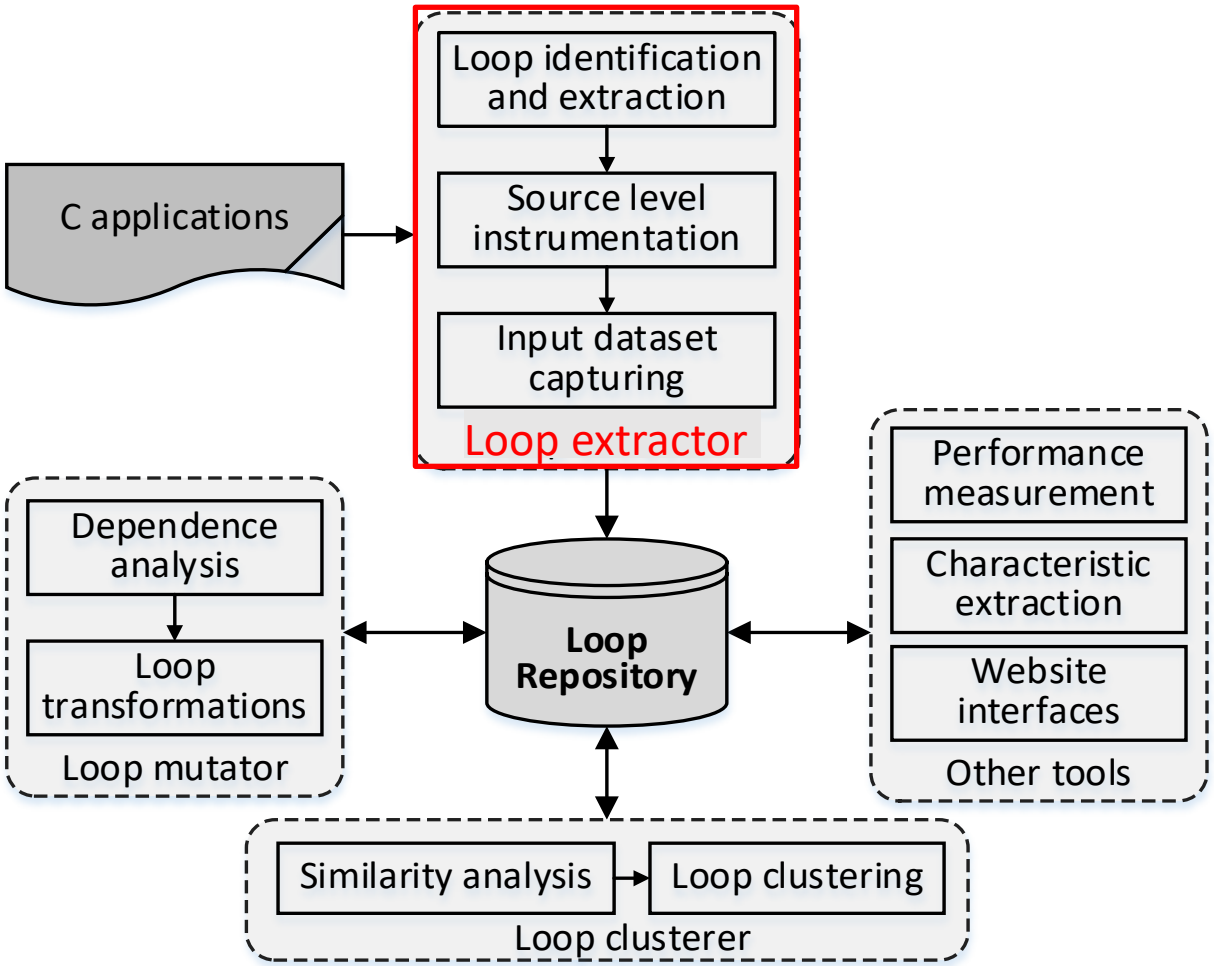
- Data are required for measurements
  - Different classes of code and target machines
  - Compilers across successive generations
  - Solo/compound transformations
- This data is not widely available
- Loop repository (https://vectorization.computer)
  - Experimentation and comparison of results
  - Effect of transformations
  - Execution time of different transformation sequences

System infrastructure

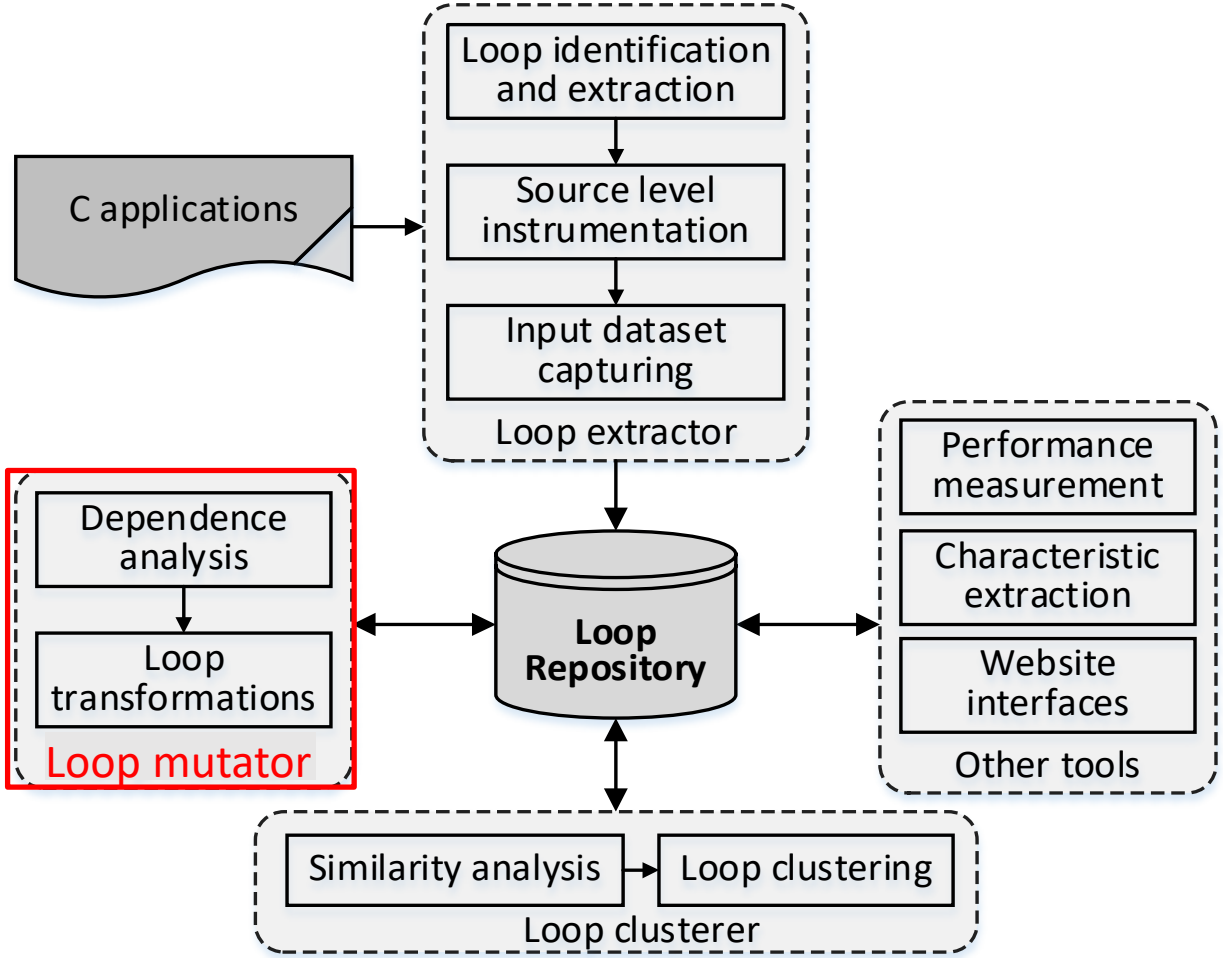
System Modules



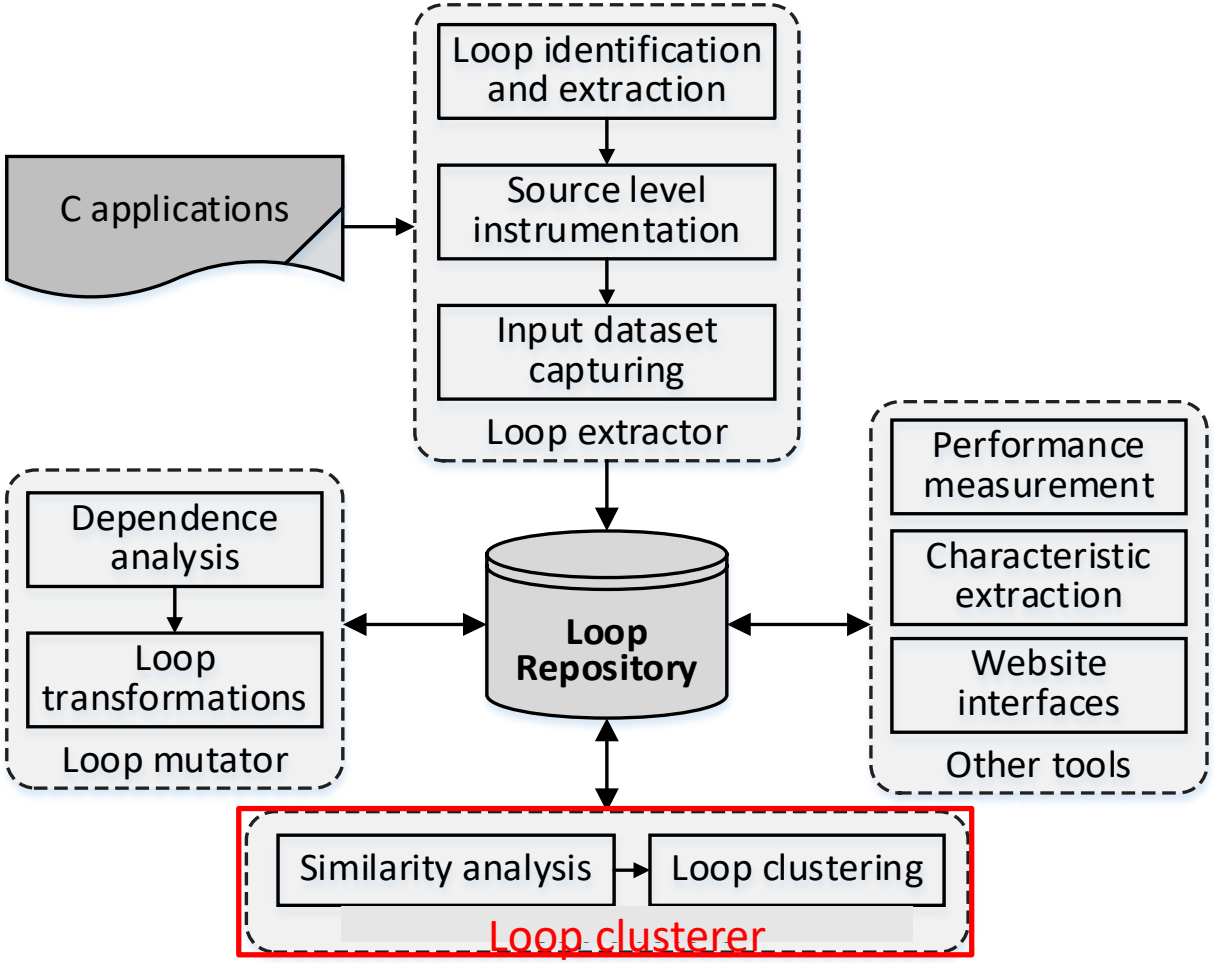
System Modules



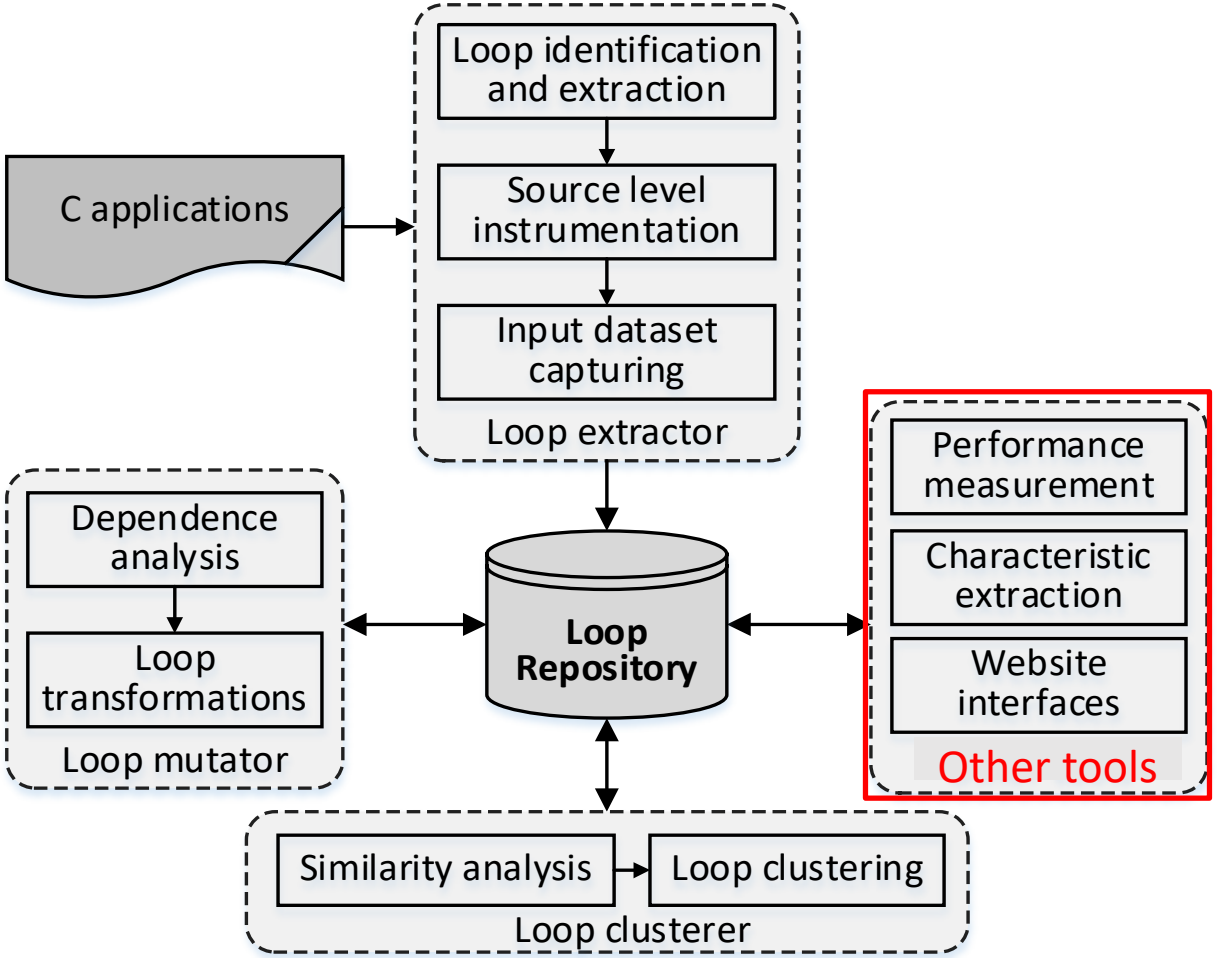
System Modules



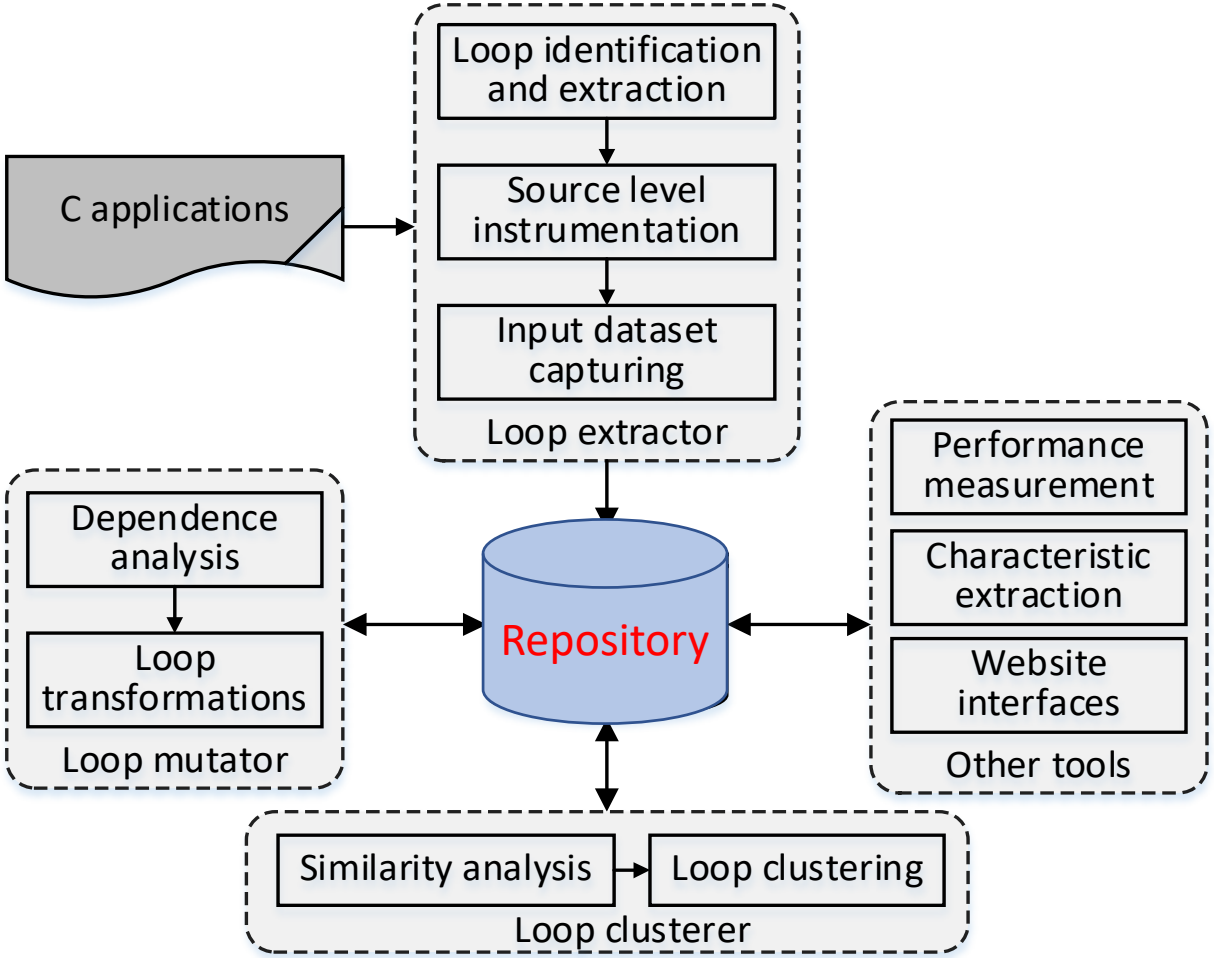
System Modules



System Modules



System Modules



System Module: Extractor

- Goal
  - Build standalone codelets with qualified loops
  - Replayable with the original data
- Based on ROSE source-to-source compiler
  - Identify interesting nodes by traversing AST
  - Insert source level instrumentation

System Module: Extractor

```
// Code before the loop
for (i = 0; i < N; i++) {
   for (j = 0; j < N; j++) {
      a[i][j] += u1[i] * v1[j] + u2[i] * v2[j];
   }
}
// Code after the loop
```

System Module: Extractor

```
// Code before the loop
for (i = 0; i < N; i++) {
   for (j = 0; j < N; j++) {
      a[i][j] += u1[i] * v1[j] + u2[i] * v2[j];
   }
}
// Code after the loop
```

System Module: Extractor

```
// Code before the loop
for (i = 0; i < N; i++) {
   for (j = 0; j < N; j++) {
      a[i][j] += u1[i] * v1[j] + u2[i] * v2[j];
   }
}
// Code after the loop
```

Data copying

Collect values for N, a, u1, v1, u2, v2
Save them to the disk

```
for (i = 0; i < N; i++) {
   for (j = 0; j < N; j++) {
      a[i][j] += u1[i] * v1[j] + u2[i] * v2[j];
   }
}
```

System Module: Extractor

```
// Code before the loop
for (i = 0; i < N; i++) {
   for (j = 0; j < N; j++) {
      a[i][j] += u1[i] * v1[j] + u2[i] * v2[j];
   }
}
// Code after the loop
```

Data copying

Collect values for N, a, u1, v1, u2, v2
Save them to the disk

```
for (i = 0; i < N; i++) {
   for (j = 0; j < N; j++) {
      a[i][j] += u1[i] * v1[j] + u2[i] * v2[j];
   }
}
```

Codelet driver

main.c

```
int i, N, j;
double A[2000][2000], u1[2000] ;
double v1[2000] , u2[2000], v2[2000] ;
Loop() {
   for (i = 0; i < N; i++) {
      for (j = 0; j < N; j++) {
         a[i][j] += u1[i] * v1[j] + u2[i] * v2[j];
      }
   }
}
```

Loop.c

Codelet driver

- Restore execution environment
- Initialize variables with collected data
- Invoke the loop
- Collect measurements
  - Performance counter values and execution cycles

Codelet: Dynamic Features

- Categories
  - memory traffic
  - memory hits
  - line fill buffer (LFB) occupancy
  - TLB related events
  - resource stalls
  - Prefetching
  - Instruction mix (PIN tool)
    - Total # of insts, total # of f.p. insts, # of SSE, AVX, and AVX2 insts, integer SSE and AVX2
    - # scalar and vector loads, # scalar and vector stores

Codelet: Dynamic Features

- Categories
  - memory traffic
  - memory hits
  - line fill buffer (LFB) occupancy
  - TLB related events
  - resource stalls
  - Prefetching
  - Instruction mix (PIN tool)
    - Total # of insts, total # of f.p. insts, # of SSE, AVX, and AVX2 insts, integer SSE and AVX2
    - # scalar and vector loads, # scalar and vector stores

From performance counters

Codelet: Dynamic Features

- Categories
  - memory traffic
  - memory hits
  - line fill buffer (LFB) occupancy
  - TLB related events
  - resource stalls
  - Prefetching
  - Instruction mix (PIN tool)
    - Total # of insts, total # of f.p. insts, # of SSE, AVX, and AVX2 insts, integer SSE and AVX2
    - # scalar and vector loads, # scalar and vector stores

Usage of features:

Codelet: Dynamic Features

- Categories
  - memory traffic
  - memory hits
  - line fill buffer (LFB) occupancy
  - TLB related events
  - resource stalls
  - Prefetching
  - Instruction mix (PIN tool)
    - Total # of insts, total # of f.p. insts, # of SSE, AVX, and AVX2 insts, integer SSE and AVX2
    - # scalar and vector loads, # scalar and vector stores

Usage of features:

•Understand performance results

Codelet: Dynamic Features

- Categories
  - memory traffic
  - memory hits
  - line fill buffer (LFB) occupancy
  - TLB related events
  - resource stalls
  - Prefetching
  - Instruction mix (PIN tool)
    - Total # of insts, total # of f.p. insts, # of SSE, AVX, and AVX2 insts, integer SSE and AVX2
    - # scalar and vector loads, # scalar and vector stores

Usage of features:

•Understand performance results

•Determine performance effect produced by our mutator

System Module: Mutator

- Goal: to search in a loop transformation space for performance improvement
- Sequence of source-to-source transformations, including:
  - Interchange
  - Tiling
  - Unrolling
  - Unroll-and-jam
  - Distribution
- Validated by dependence analysis (if necessary)

System Module: Clusterer

- Goals: reduce redundant loops and identify loop patterns
- For each codelet, we collect:
  - Data for 28 different static features
  - Dependence graph
- Uses a simple, hierarchical clustering algorithm
- Distance metric is weighted
  - Favors the dependence graph when it is available

- loop nest level
- lower/upper bounds and stride
- # statements
- # memory operations
- # floating-point/integer operations
- # constant
- # branches
- # basic blocks (BB) and edges in the loop's CFG

The Repository: LORE

- Statistics (as of the writing of the paper)
  - 25 benchmark suites
    - Audio/Video codecs, deep learning, SPEC, real-world applications from GitHub, etc.
  - ~2500 codelets, ~90,000 mutated codelet versions
  - 40 performance counters
  - Instruction mix
  - 28 static features
  - 3 compilers each with 2 versions
    - 5 vectorization settings on the same HW configuration
  - These numbers keep growing

Platform

- Software

GCC	ICC	Clang
4.8.5	15.0.6	3.6.2
6.2.0	17.0.1	4.0.0

- Hardware

Parameters	Value
Processor Model	Intel Xeon E5-1630 v3
L1 cache	32KB
L2 cache	256KB
L3 cache	10MB
Operating system	Ubuntu 16.04

Compiler flags

GCC 6.2.0	Baseline	-O3 -ftree-vectorize -ffast-math -funsafe-loop-optimizations -ftree-loop-if-convert-stores -march=native -mtune=native
	SSE	[Baseline Flags] -fvect-cost-model=unlimited -mno-avx -mno-avx2
	AVX	[Baseline Flags] -fvect-cost-model=unlimited -mno-avx2
	AVX2	[Baseline Flags] -fvect-cost-model=unlimited
	Scalar	-O3 -fno-tree-vectorize -ffast-math -funsafe-loop-optimizations -march=native -mtune=native

Clang 4.0.0	Baseline	-O3 -fvectorize -fslp-vectorize -fslp-vectorize-aggressive -ffast-math -march=native -mtune=native
	SSE	[Baseline Flags] -mno-avx -mno-avx2
	AVX	[Baseline Flags] -mno-avx2
	AVX2	[Baseline Flags]
	Scalar	-O3 -fno-vectorize -fno-slp-vectorize -ffast-math -march=native -mtune=native

ICC 16.0.3	Baseline	-O3 -restrict -xHost -ipo
	SSE	-O3 -restrict -ipo -vec-threshold0 -msse4.2
	AVX	-O3 -restrict -ipo -vec-threshold0 -mavx
	AVX2	[Baseline Flags] -vec-threshold0
	Scalar	[Baseline Flags] -no-vec

Website (https://vectorization.computer)

- Publicly accessible
- Users can
  - View/download source/assembly
  - Discuss loops
- Visualize results
- Run queries

Benchmark: NPB 2.3-OpenACC-C
Application: BT

File: bt.c
Function: add
Line: 179

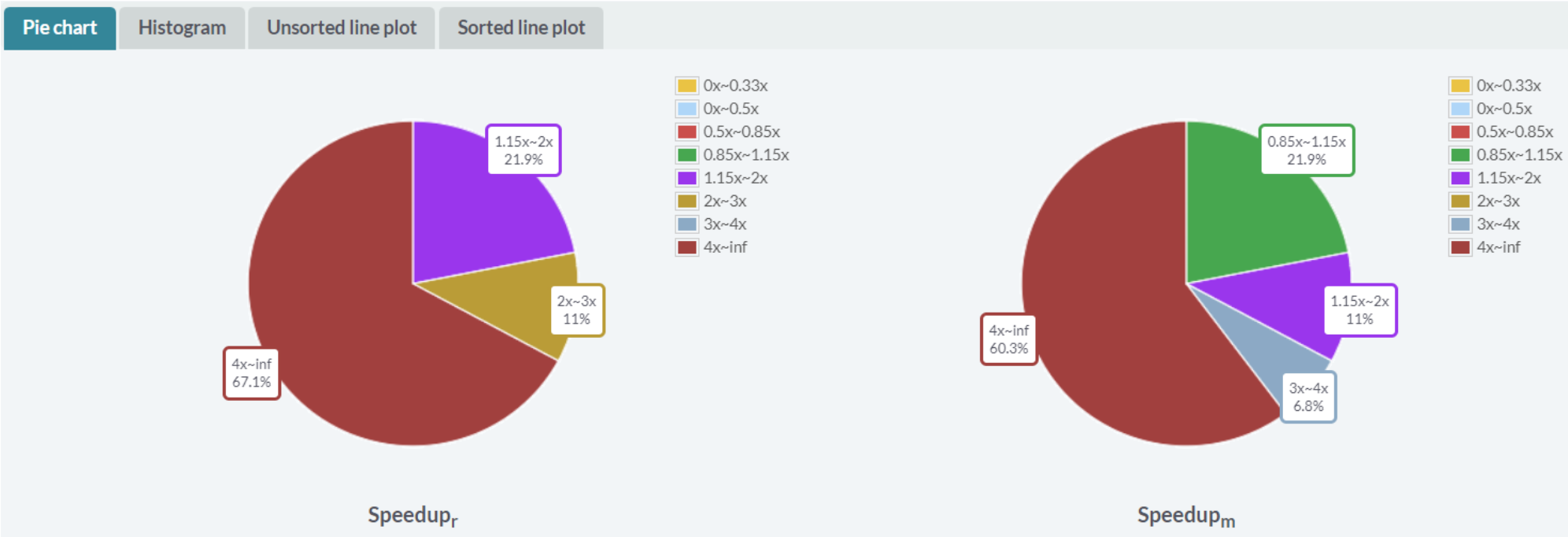
Baseline source code

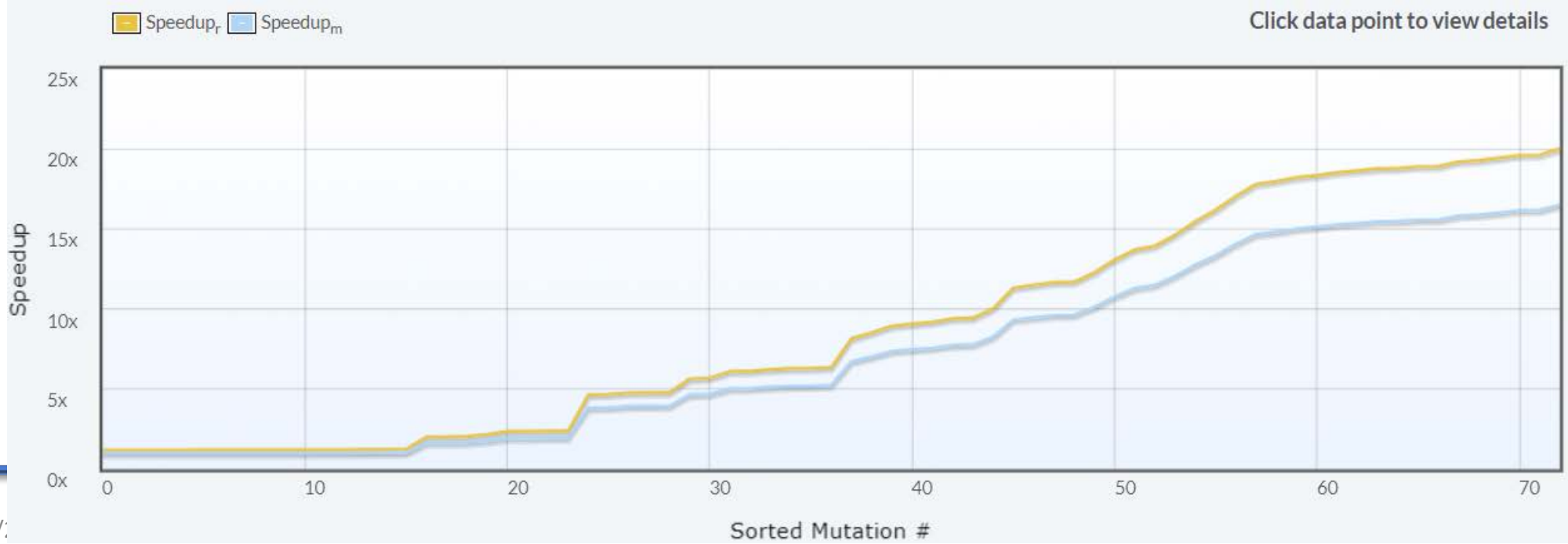
Processed baseline | Original baseline | Download

```
#include <inttypes.h>
#include <stdio.h>
#include <stdlib.h>
extern int i;
extern int grid_points[3];
extern int j;
extern int k;
extern int m;
extern double u[(64 + 1) / 2 * 2 + 1][(64 + 1) / 2 * 2 + 1][(64 + 1) / 2 * 2 + 1][5];
extern double rhs[64 / 2 * 2 + 1][64 / 2 * 2 + 1][64 / 2 * 2 + 1][5];

void loop()
{
  int __arr_sub_0__ = grid_points[0];
  int __arr_sub_1__ = grid_points[1];
  int __arr_sub_2__ = grid_points[2];
  int __i_3__ = i;
  int __j_4__ = j;
  int __k_5__ = k;
  int __m_6__ = m;

#pragma scop
{
    for (__i_3__ = 1; __i_3__ <= __arr_sub_0__ - 1 - 1; __i_3__ += 1) {
      for (__j_4__ = 1; __j_4__ <= __arr_sub_1__ - 1 - 1; __j_4__ += 1) {
        for (__k_5__ = 1; __k_5__ <= __arr_sub_2__ - 1 - 1; __k_5__ += 1) {
          for (__m_6__ = 0; __m_6__ <= 4; __m_6__ += 1) {
            u[__i_3__][__j_4__][__k_5__][__m_6__] = u[__i_3__][__j_4__][__k_5__][__m_
```

Static features of processed baseline

Nest levels	4
Statements	13
Memory accesses	3
Floating point binary operations	5
Integer binary operations	18
Integer unary operations	0
Assignments	9
Constants	15
Constant 0s	1
Constant 1s	13
Binary operations with integer constants	15
Basic blocks	14
Basic blocks with 1 successor	10
Basic blocks with 2 successors	4
Basic blocks with 3 or more successors	0
Basic blocks with 1 predecessor	9
Basic blocks with 2 predecessors	4


Pie chart
Histogram
Unsorted line plot
Sorted line plot
0x~0.33x
0x~0.5x
0.5x~0.85x
0.85x~1.15x
1.15x~2x
2x~3x
3x~4x
4x~inf
1.15x~2x 21.9%
2x~3x 11%
4x~inf 67.1%
Speedup$_r$
0.85x~1.15x 21.9%
1.15x~2x 11%
3x~4x 6.8%
4x~inf 60.3%
Speedup$_m$


Speedup$_r$
Speedup$_m$
Click data point to view details
Speedup
25x
20x
15x
10x
5x
0x
0
10
20
30
40
50
60
70
Sorted Mutation #

Hardware Dynamic Counters

Show 10 entries Search:

Metric	Reference	Scalar	SSE	AVX	AVX2
INST_RETIRED_ANY	21325	21325	19798	18673	16213
CPU_CLK_UNHALTED_THREAD	12229	11508	8817	8285	7791
MEM_LOAD_UOPS_RETIRED_L1_HIT_PS	8685	8674	7053	7072	6862
L1D_PEND_MISS_PENDING	790	677	586	592	601
L1D_PEND_MISS_PENDING_CYCLES	730	531	458	514	495
RESOURCE_STALLS_ANY	946	1043	39	178	153
RESOURCE_STALLS_MEM_RS	1008	1010	27	169	183
RESOURCE_STALLS_SB	940	1019	18	141	39
L1D_REPLACEMENT	158	163	104	102	113
RESOURCE_STALLS_RS	6	4	27	85	136

Showing 1 to 10 of 40 entries Previous 1 2 3 4 Next

Dynamic Instruction Counts

Show 10 entries Search:

Metric	Reference	Scalar	SSE	AVX	AVX2
inst_cnt	20960	20960	19433	18308	15848
fp_arith_inst_cnt	5000	5000	6000	6000	3000
fp_sse_inst_cnt	0	0	4000	0	0
vector_inst_exclude_mem_cnt	0	0	4000	4000	2000
scalar_ld_cnt	8697	8697	3702	3702	3702

Advanced

Online view | Download CSV

SELECT benchmark, application, file, line, n_mutations, transformations FROM loops limit 100 Run

Show 25 entries Search:

benchmark	application	file	line	n_mutations	transformations
ALPBench	MPGenc	predict.c	511	4	interchange tiling unrolling distribution
ALPBench	Ray_trace	imageio.c	167	4	interchange tiling unrolling distribution
ALPBench	csuFaceIdEval_5.1	csuCommonMatrix.c	435	4	interchange tiling unrolling distribution
ASC-llnl	amgmk	csr_matvec.c	124	4	interchange tiling unrolling distribution
cortexsuite	svd3	svd.c	293	4	interchange tiling unrolling distribution
Fhourstones	default	c4.c	131		interchange tiling unrolling distribution
FreeBench	neural	neural.c	711	16	interchange tiling unrolling distribution
FreeBench	pifft	fftsg.c	653	4	interchange tiling unrolling distribution
FreeBench	pifft	pifft.c	558	16	interchange tiling unrolling distribution
GitApps	gap	trans.c	520		interchange tiling unrolling distribution
GitApps	mozjpeg	jchuff.c	1009	4	interchange tiling unrolling distribution

Sample usage

Comparison of vectorizers

	GCC 6.2.0	ICC 17.0.1	Clang 4.0.0
# of loops included	959		
# (%) of loops originally having effective vectorization	273 (28.5%)	395 (41.2%)	216 (22.5%)
# (%) of additional loops vectorized effectively and beneficial against the original baseline after mutation	77 (8.0%)	63 (6.6%)	98 (10.2%)
Subtotal	350 (36.5%)	458 (47.8%)	314 (32.7%)

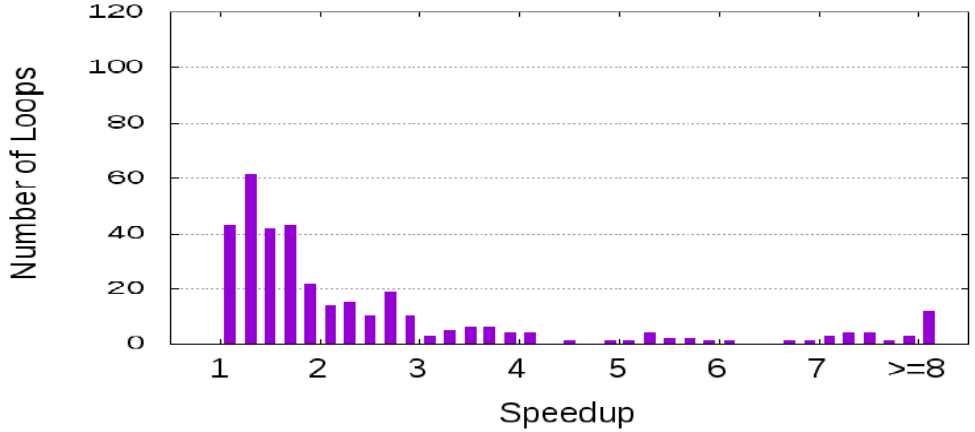
- Effective/beneficial means at least 1.15x speedup

Comparison of vectorizers

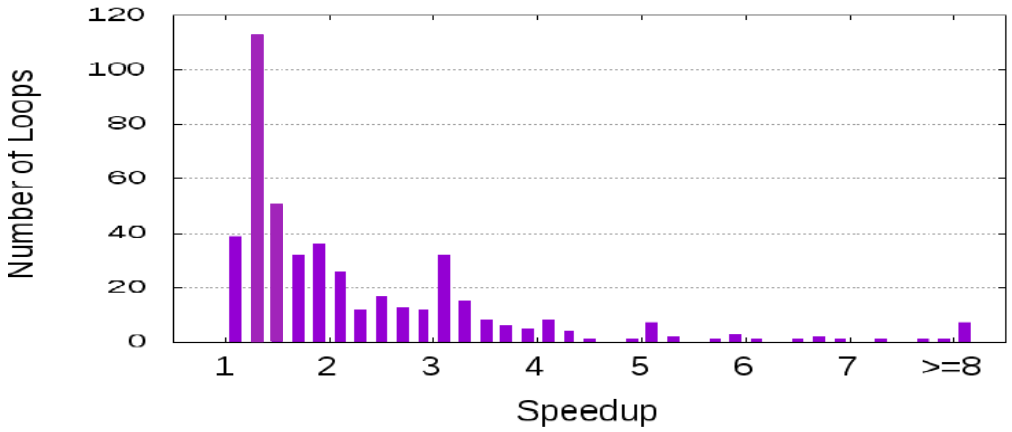
	GCC 6.2.0	ICC 17.0.1	Clang 4.0.0
# of loops included	959		
# (%) of loops originally having effective vectorization	273 (28.5%)	395 (41.2%)	216 (22.5%)
# (%) of additional loops vectorized effectively and beneficial against the original baseline after mutation	77 (8.0%)	63 (6.6%)	98 (10.2%)
Subtotal	350 (36.5%)	458 (47.8%)	314 (32.7%)

- Effective/beneficial means at least 1.15x speedup

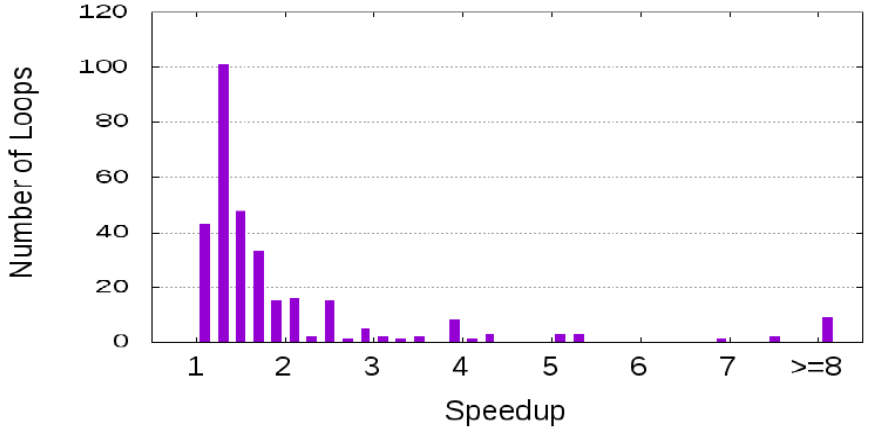
Comparison of vectorizers



(a) GCC

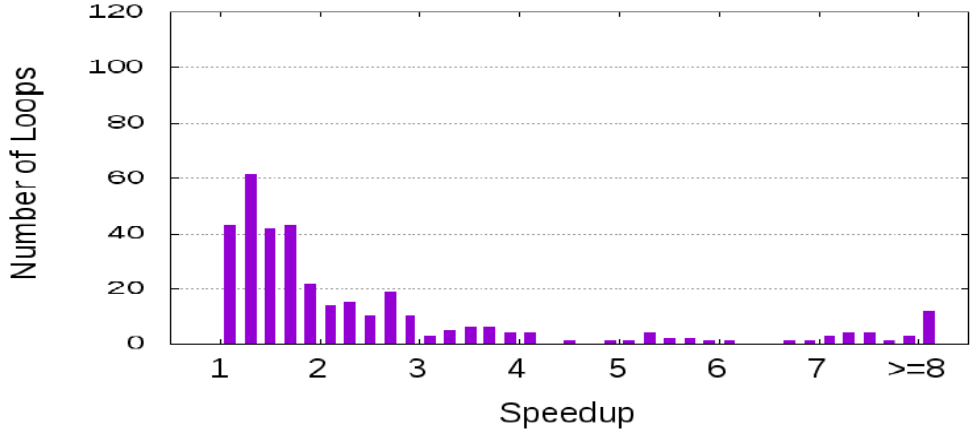


(b) ICC

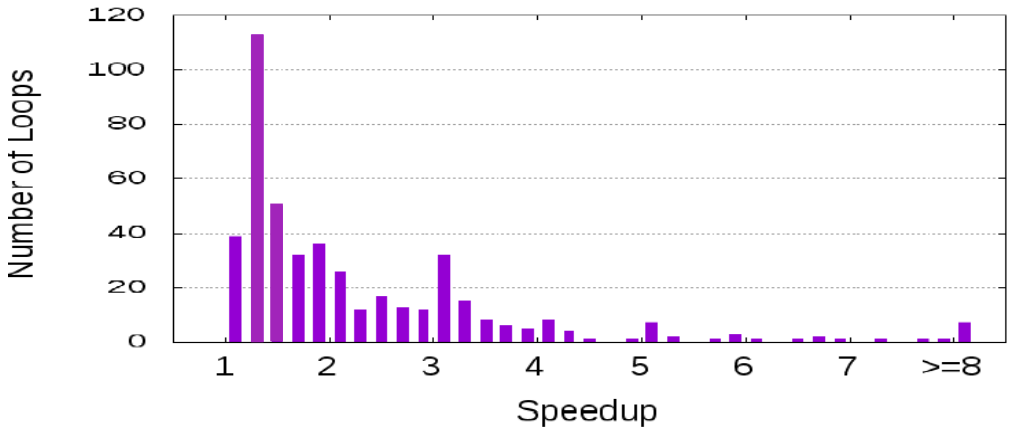


(c) Clang

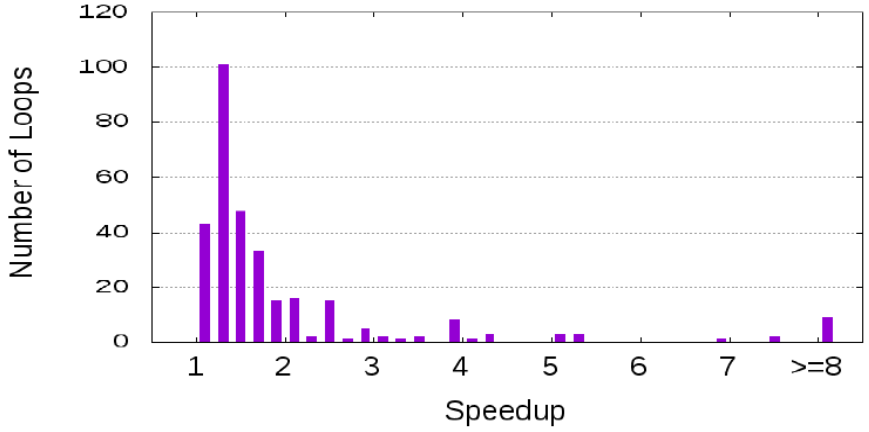
Comparison of vectorizers



(a) GCC

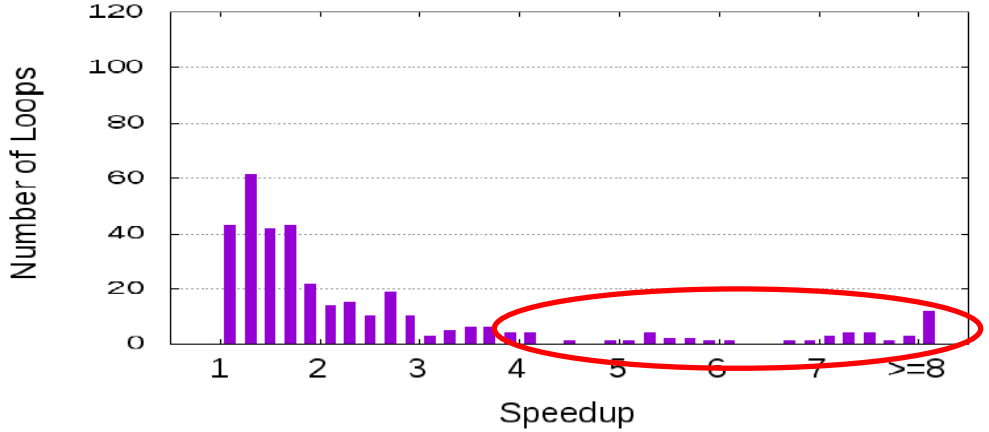


(b) ICC

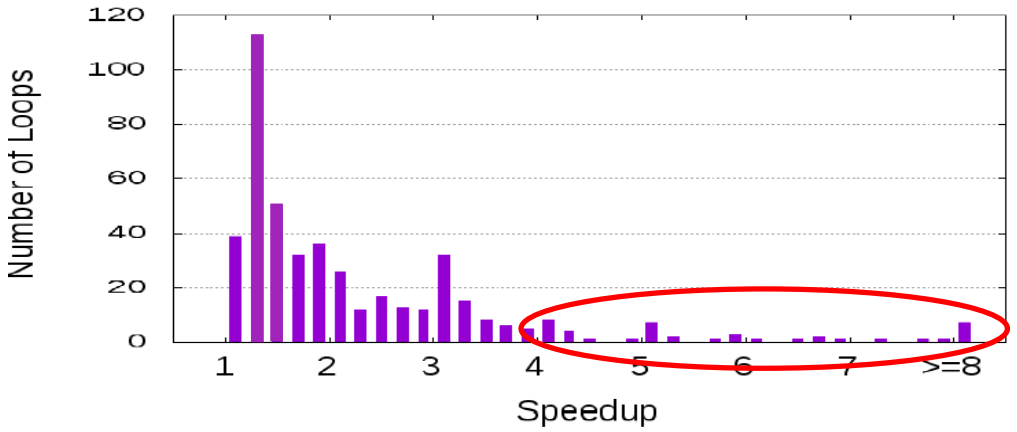


(c) Clang

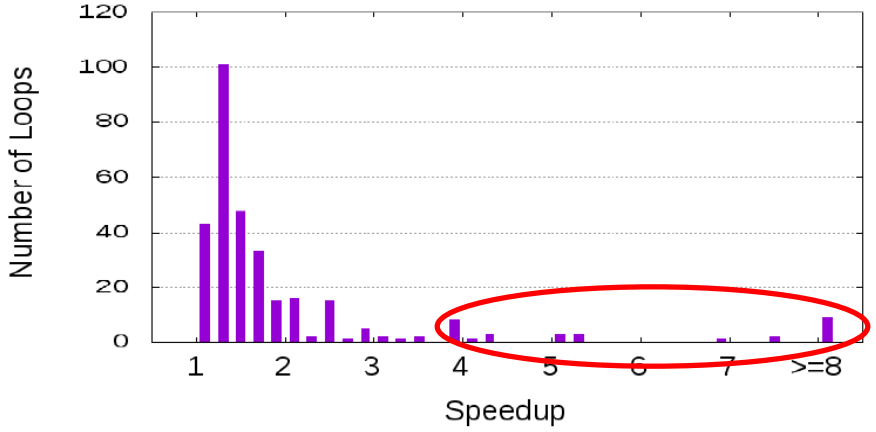
Comparison of vectorizers



(a) GCC

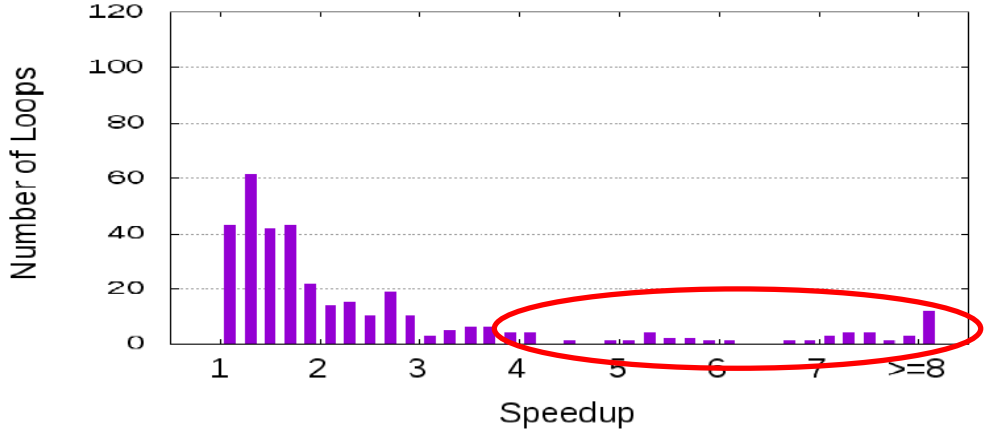


(b) ICC

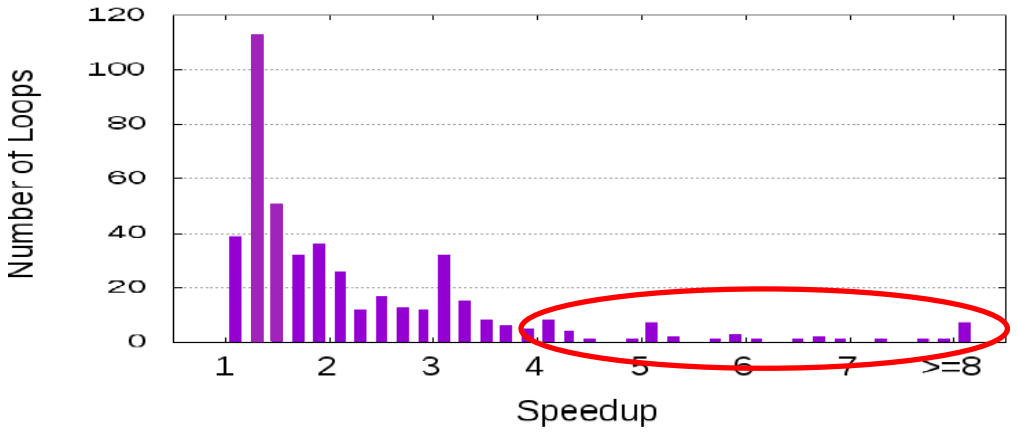


(c) Clang

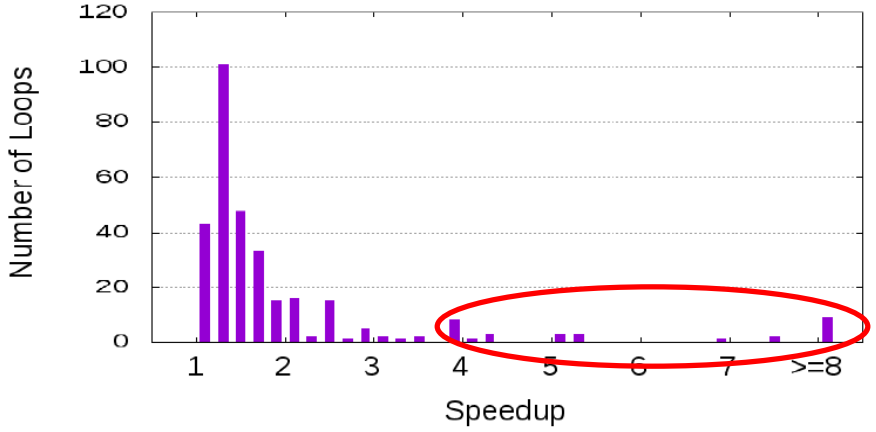
Comparison of vectorizers



(a) GCC

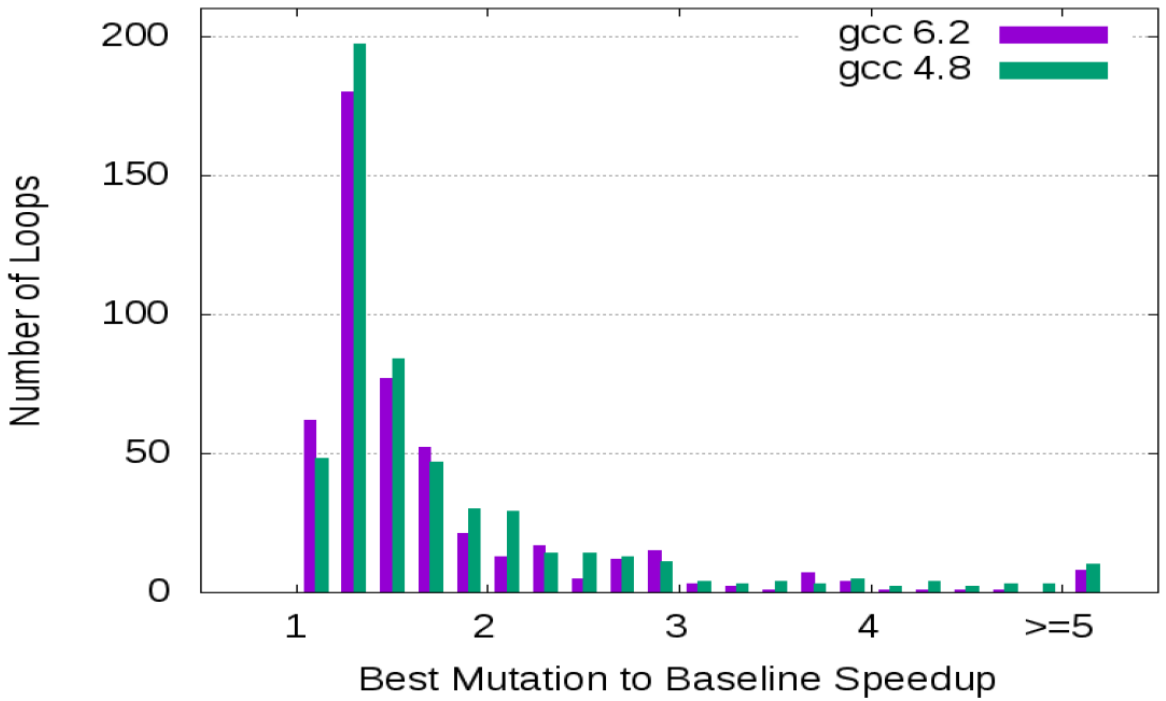


(b) ICC

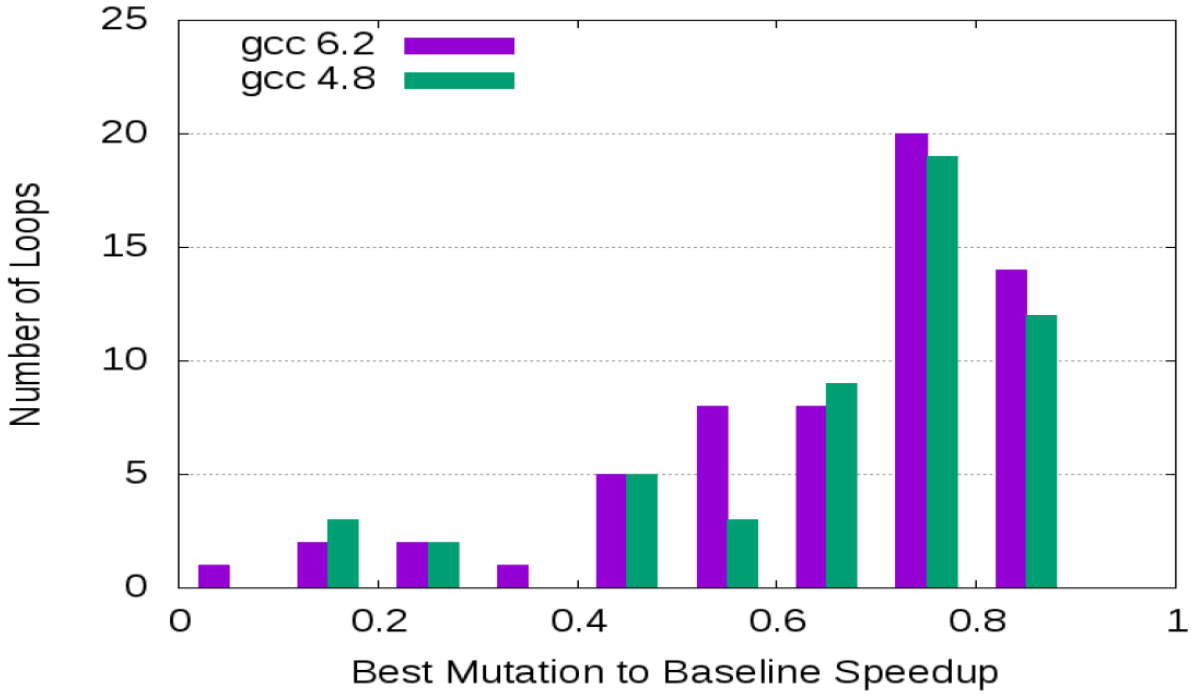


(c) Clang

GCC 6.2.0 VS GCC 4.8.5

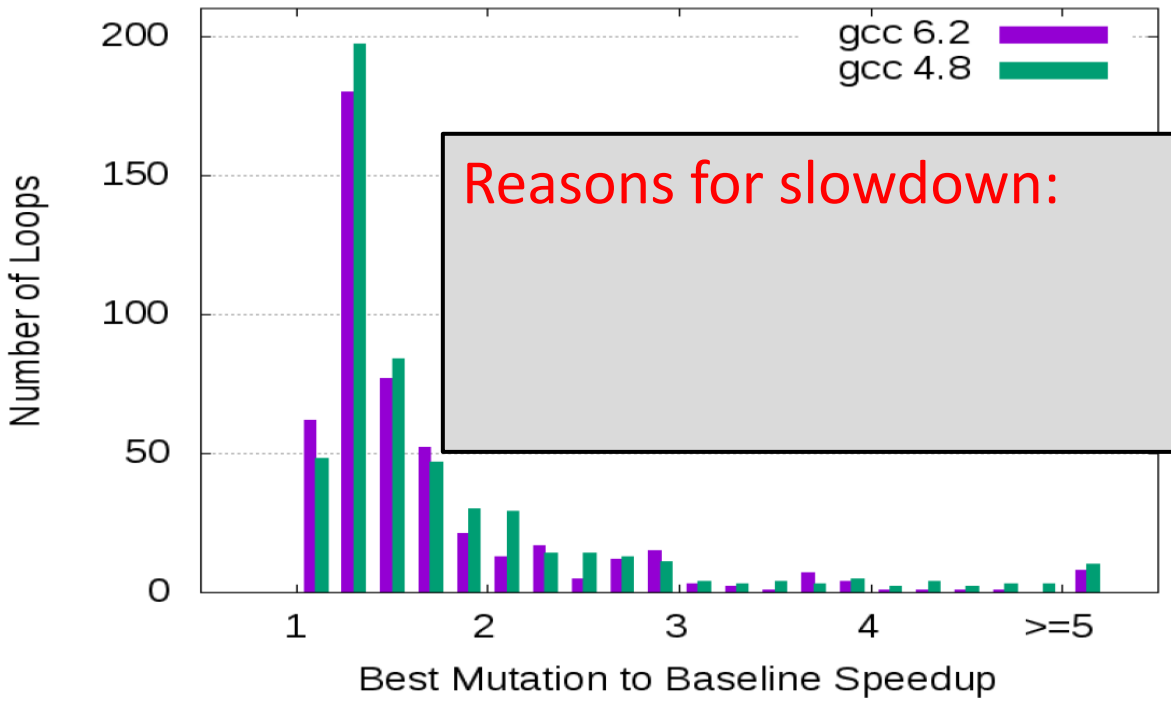


beneficial mutations (speedup >= 1.15)

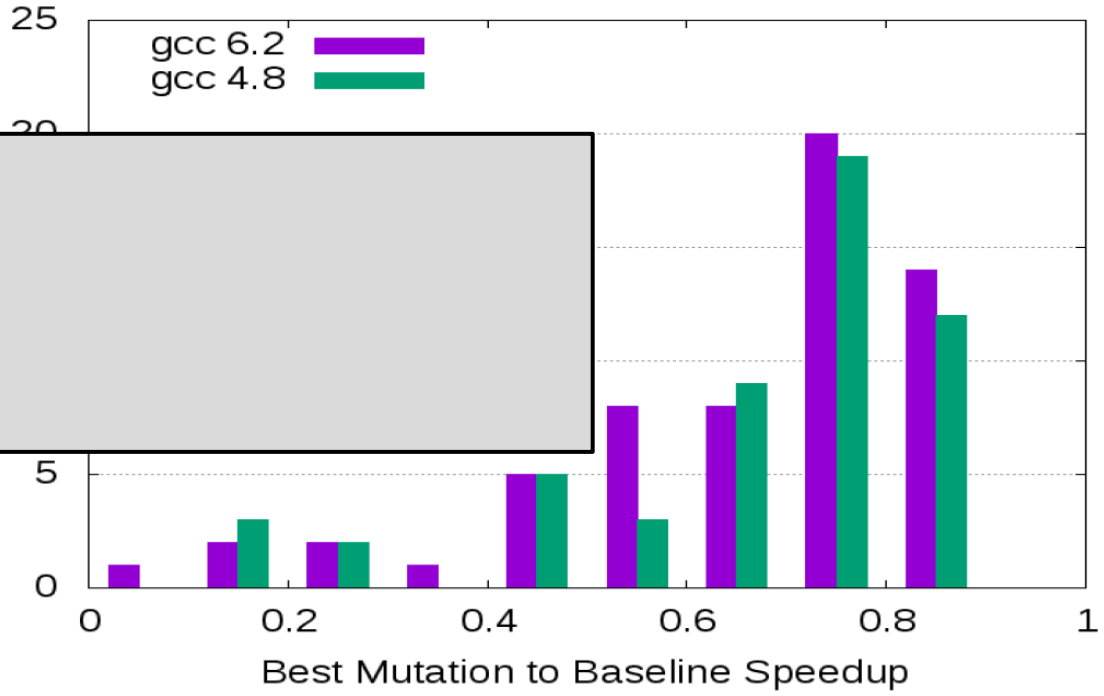


unfavorable mutations (slowdown > 1.15)

GCC 6.2.0 VS GCC 4.8.5



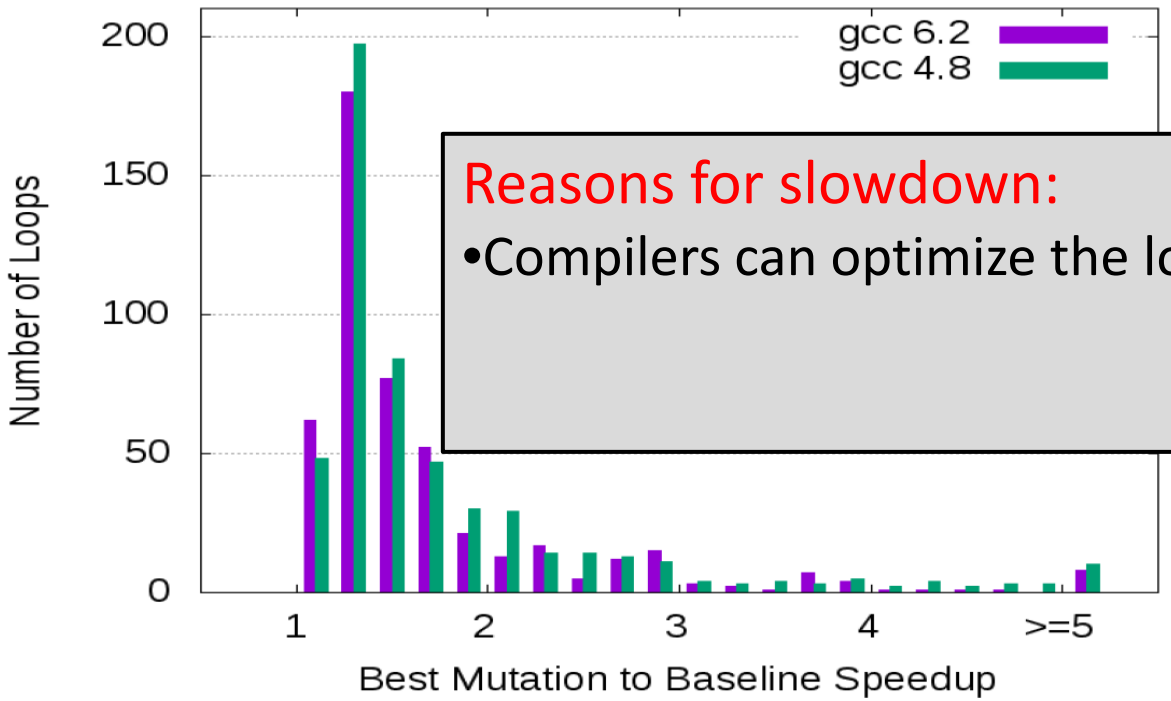
beneficial mutations (speedup >= 1.15)



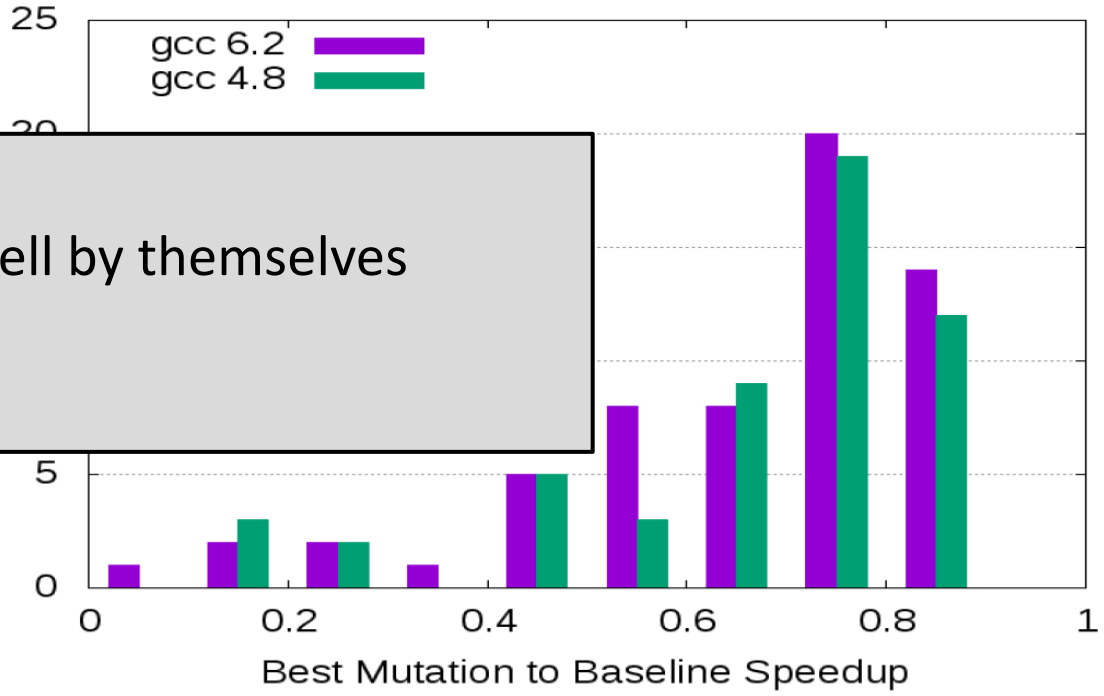
unfavorable mutations (slowdown > 1.15)

Reasons for slowdown:

GCC 6.2.0 VS GCC 4.8.5



beneficial mutations (speedup >= 1.15)

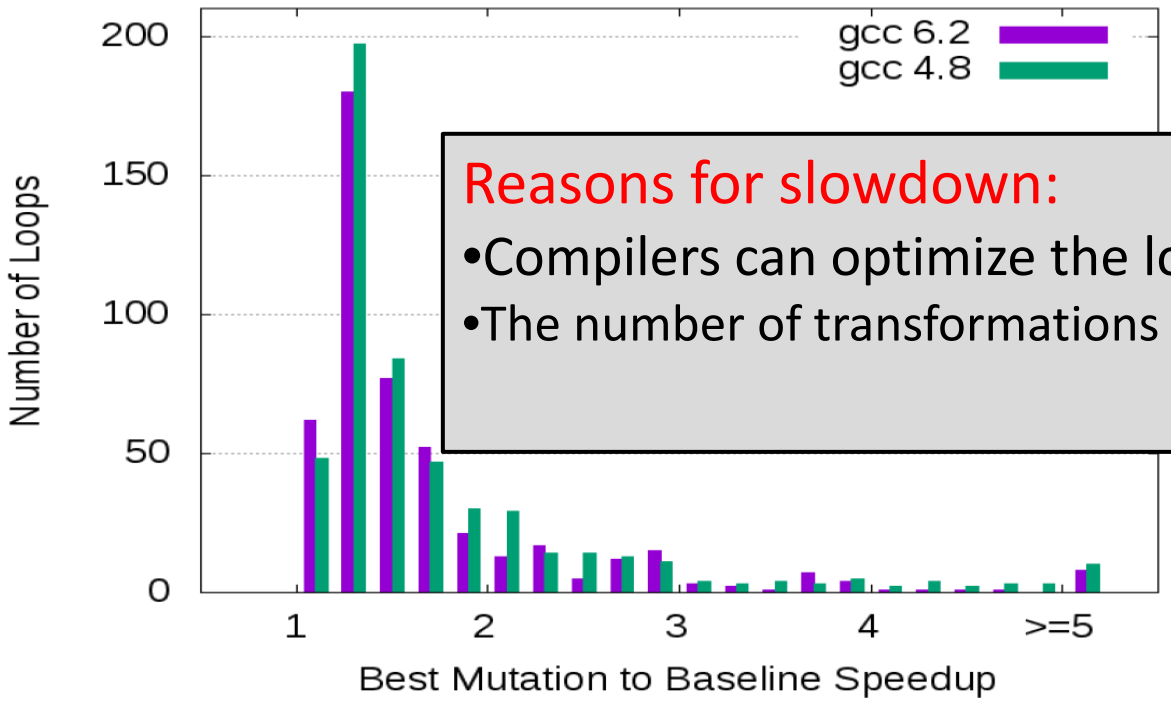


unfavorable mutations (slowdown > 1.15)

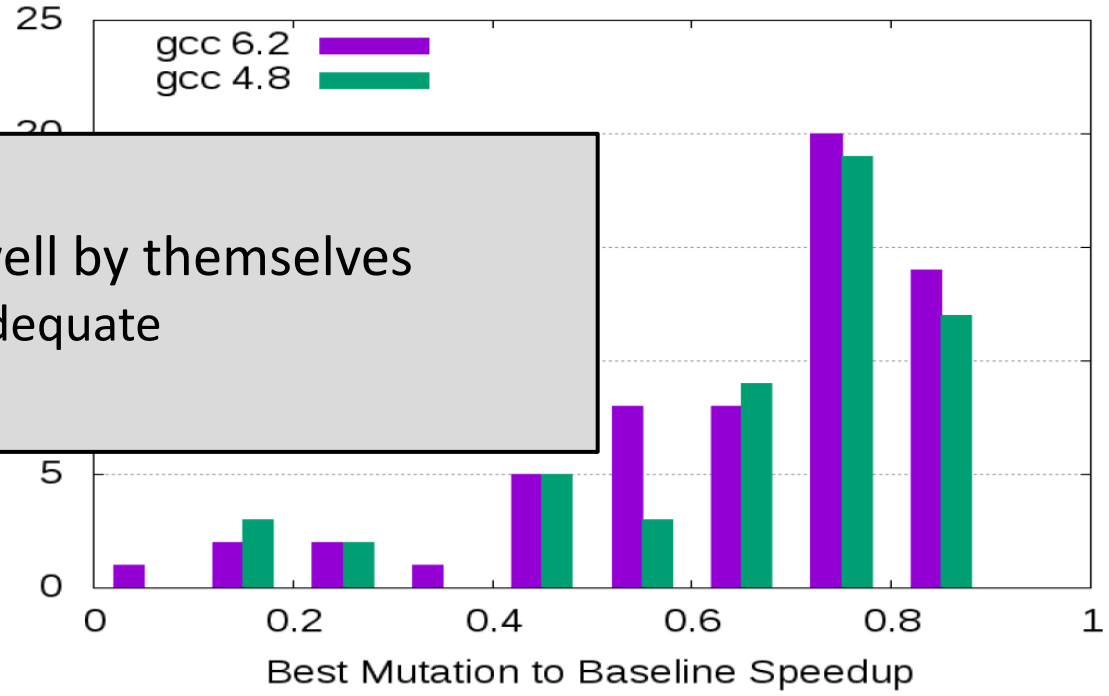
Reasons for slowdown:

- Compilers can optimize the loop well by themselves

GCC 6.2.0 VS GCC 4.8.5



beneficial mutations (speedup >= 1.15)



unfavorable mutations (slowdown > 1.15)

Reasons for slowdown:

- Compilers can optimize the loop well by themselves
- The number of transformations is inadequate

Conclusion

- Developed a loop repository which consists of a series of tools to extract, mutate, and classify loops
- Provided a broad range of characteristics for each codelet
- The repository could help the community to identify the strength and weakness of a compiler
- Sample usage showed that there is still room for compilers to improve

Future work

- Incorporate more loop transformations
- Measure the importance of loops during extraction
- Evaluate more compilers and more platforms


ANY
QUESTIONS?