EUROPEAN LLVM DEVELOPERS' MEETING 2019

DOE PROXY APPS: COMPILER PERFORMANCE ANALYSIS AND OPTIMISTIC ANNOTATION EXPLORATION

BRIAN HOMERDING
ALCF
Argonne National Laboratory
ECP Proxy Apps

JOHANNES DOERFERT
ALCF
Argonne National Laboratory

April 9th, 2019
Brussels, Belguim

UCHICAGO ARGONNE LLC
U.S. DEPARTMENT OF ENERGY
Argonne National Laboratory is a U.S. Department of Energy laboratory managed by UChicago Argonne, LLC.

OUTLINE

- Context (Proxy Applications)
- HPC Performance Analysis & Compiler Comparison
- Modelling Math Function Memory Access
- Information and the Compiler
- Optimistic Annotations
- Optimistic Suggestions

ECP PROXY APPLICATION PROJECT

Co-Design

- Improve the quality of proxies
- Maximize the benefit received from their use

ECP PathForward

Team

- David Richards, LLNL (Lead PI)
- Hal Finkel, ANL
- Thomas Uram, ANL
- Brian Homerding, ANL
- Christoph Junghans, LANL
- Robert Pavel, LANL
- Vinay Ramakrishnaiah, LANL
- Peter McCorquodale, LBL
- Abhinav Bhatele, LLNL
- Nikhil Jain, LLNL
- Bronson Messer, ORNL
- Tiffany Mintz, ORNL
- Shirley Moore, ORNL
- Jeanine Cook, SNL
- Courtenay Vaughan, SNL

Proxy Applications are used by Application Teams, Co-Design Centers, Software Technology Projects and Vendors

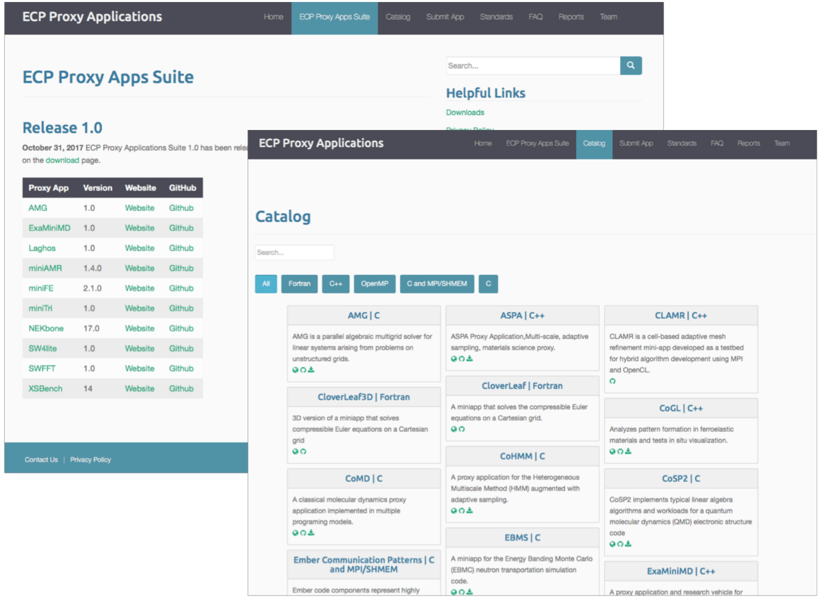
PROXY APPLICATIONS

– Proxy applications are models for one or more features of a parent application

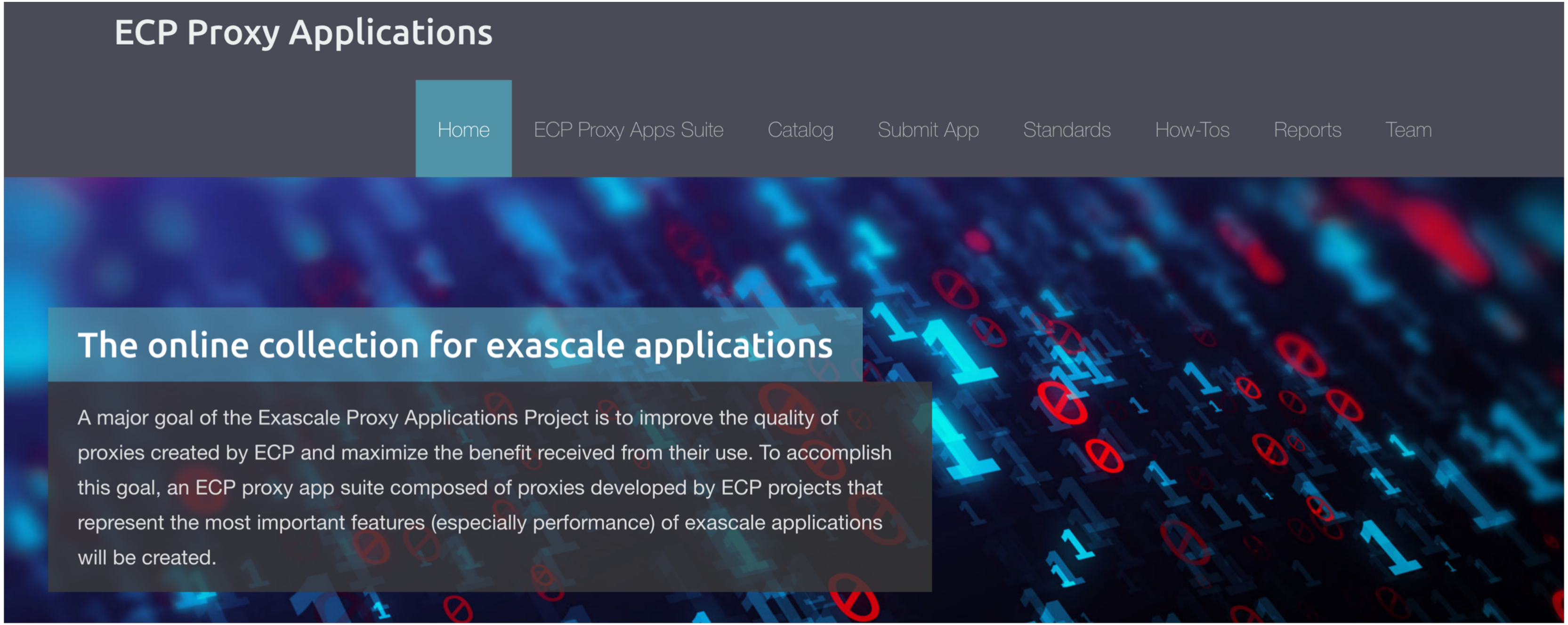
– Can model different parts
  - Performance critical algorithm
  - Communication patterns
  - Programming models

– Come in different sizes
  - Kernels
  - Skeleton apps
  - Mini apps

https://proxyapps.exascaleproject.org



ECP PROXY APPLICATION PROJECT



WHY LOOK AT PROXY APPS

- Proxy applications aim to hit a balance of complexity and usability
- Represent the performance critical sections of HPC code
- Often have various versions (MPI, OpenMP, CUDA, OpenCL, Kokkos)

Issues

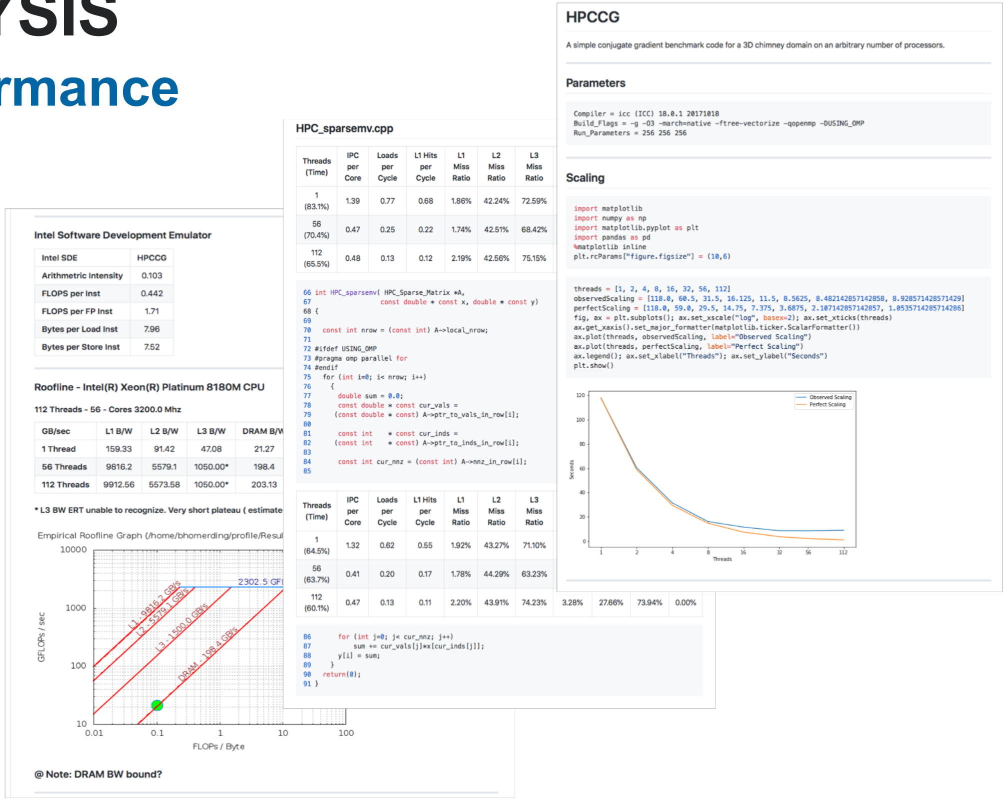
- They are designed to be experimented with, they are not benchmarks until the problem size is set
- No common test runner

HPC PERFORMANCE ANALYSIS & COMPILER COMPARISON

PERFORMANCE ANALYSIS

Quantifying Hardware Performance

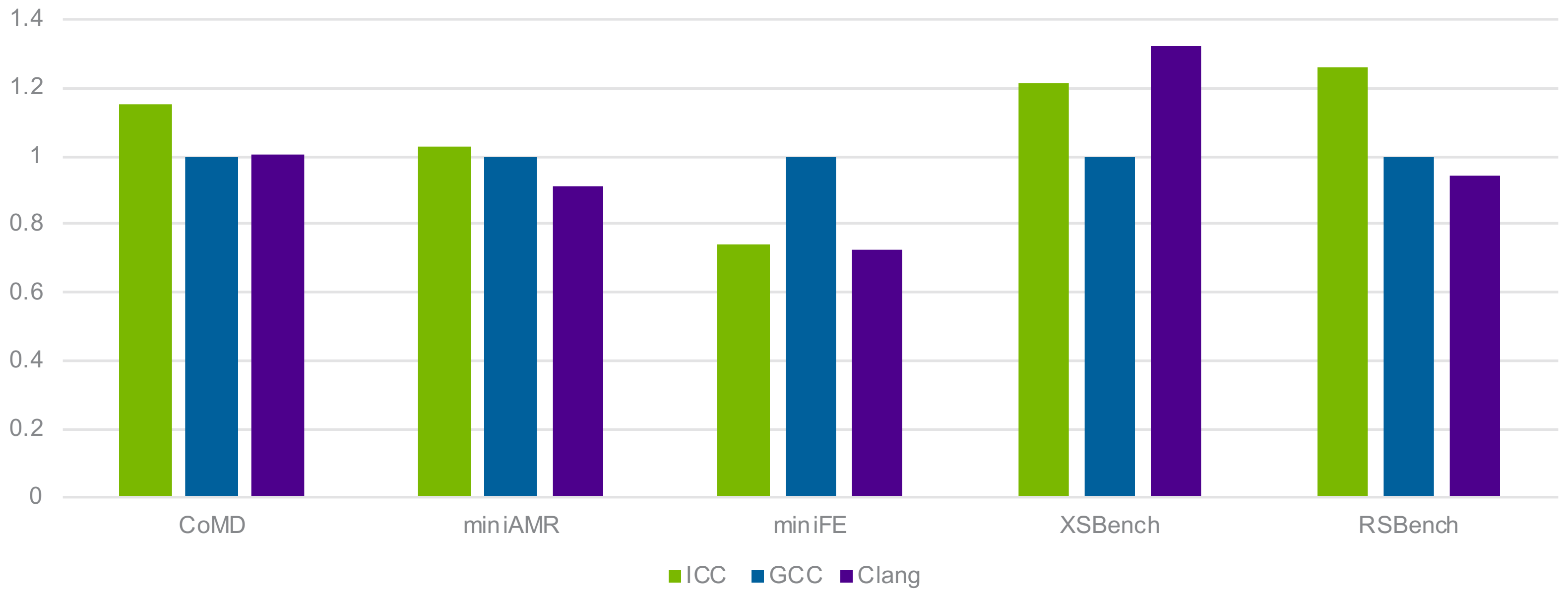
- Understand representative problem sizes
  - How to scale the problem to Exascale?
- What are the hardware characteristics of different classes of codes? (PIC, MD, CFD)
- Why is the compiler unable to optimize the code? Can we enable it to?



COMPILER FOCUS METHODOLOGY

- Get a performant version built with each compiler
- Identify room for improvement
- Collecting a wide array of hardware performance counters
- Utilize these hardware counters alongside specific code segments to identify areas where we are underperforming

RESULTS



1.4
1.2
1
0.8
0.6
0.4
0.2
0
CoMD
miniAMR
miniFE
XSBench
RSBench
ICC
GCC
Clang

RSBENCH MOTIVATING EXAMPLE

```
for( int i = 0; i < input.numL; i++ )
{
        phi = data.pseudo_K0RS[nuc][i] * sqrt(E);

        if( i == 1 )
                phi -= - atan( phi );
        else if( i == 2 )
                phi -= atan( 3.0 * phi / (3.0 - phi*phi));
        else if( i == 3 )
                phi -= atan(phi*(15.0-phi*phi)/(15.0-6.0*phi*phi));

        phi *= 2.0;

        sigTfactors[i] = cos(phi) - sin(phi) * _Complex_I;
}
```

GENERATED ASSEMBLY

Clang

```
callq   cos
vmovsd  %xmm0, 8(%rsp)                # 8-byte Spill
vmovsd  56(%rsp), %xmm0               # 8-byte Reload
                                      # xmm0 = mem[0],zero
callq   sin
vmovsd  .LCPI2_4(%rip), %xmm1         # xmm1 = mem[0],zero
vmovapd %xmm1, %xmm2
```

GCC

```
addq    $1, %rbx
addq    $16, %rbp
call    sincos
vpxord  %zmm1, %zmm1, %zmm1
vmovsd  40(%rsp), %xmm0
```

MODELING MATH FUNCTION MEMORY ACCESS

DESIGN

- Handle the special case
- Model the memory access of the math functions
- Expand Support in the backend
- Expose the functionality to the developer

DESIGN

- Handle the special case
  - Combine sin() and cos() in SimplifyLibCalls
- Model the memory access of the math functions

- Expand Support in the backend

- Expose the functionality to the developer

DESIGN

- Handle the special case
  – Combine sin() and cos() in SimplifyLibCalls
- Model the memory access of the math functions
  – Mark calls that only write errno as WriteOnly
- Expand Support in the backend
- Expose the functionality to the developer

DESIGN

- Handle the special case
  - Combine sin() and cos() in SimplifyLibCalls
- Model the memory access of the math functions
  - Mark calls that only write errno as WriteOnly
- Expand Support in the backend
  - Make use of the attribute – EarlyCSE with MSSA

- Expose the functionality to the developer

DESIGN

- Handle the special case
  - Combine sin() and cos() in SimplifyLibCalls
- Model the memory access of the math functions
  - Mark calls that only write errno as WriteOnly
- Expand Support in the backend
  - Make use of the attribute – EarlyCSE with MSSA
  - Gain coverage of the attribute – Infer the attribute in FunctionAttrs
- Expose the functionality to the developer

DESIGN

- Handle the special case
  - Combine sin() and cos() in SimplifyLibCalls
- Model the memory access of the math functions
  - Mark calls that only write errno as WriteOnly
- Expand Support in the backend
  - Make use of the attribute – EarlyCSE with MSSA
  - Gain coverage of the attribute – Infer the attribute in FunctionAttrs
- Expose the functionality to the developer
  - Create an attribute in clang FE

INFORMATION AND THE COMPILER

QUESTIONS

- What information can we encode that we can’t infer?
- Does this information improve performance?
- If not, is it because the information is not useful or not used?
- How do I know what information I should add?
- How much performance is lost by information that is correct but that compiler cannot prove?

EXAMPLE

>> clang -O3

```
int *globalPtr;
void external(int*, std::pair<int>&);

int bar(uint8_t LB, uint8_t UB) {
    int sum = 0;
    std::pair<int> locP = {5, 11};
    external(&sum, locP);

    for (uint8_t u = LB; u != UB; u++)
        sum += *globalPtr + locP.first;
    return sum;
}
```

EXAMPLE

>> clang -O3

```
int *globalPtr;
void external(int*, std::pair<int>&)
__attribute__((pure));

int bar(uint8_t LB, uint8_t UB) {
  int sum = 0;
  std::pair<int> locP = {5, 11};
  external(&sum, locP);
  __builtin_assume(LB <= UB);
  for (uint8_t u = LB; u != UB; u++)
    sum += *globalPtr + locP.first; return
  sum;
}
```

EXAMPLE

>> clang -O3

```
int *globalPtr;
void external(int*, std::pair<int>&);

int bar(uint8_t LB, uint8_t UB) {
  int sum = 0;
  std::pair<int> locP = {5, 11};
  external(&sum, locP);

  return (UB - LB) * (*globalPtr + 5);
}
```

OPTIMISTIC ANNOTATIONS

IN A NUTSHELL

```
void baz(int *A);
```

>> clang -O3 ...

>> verify.sh --> Success

IN A NUTSHELL

```
void baz(__attribute__((readnone)) int *A);
```

```
>> clang -O3 ...
>> verify.sh --> Failure
```

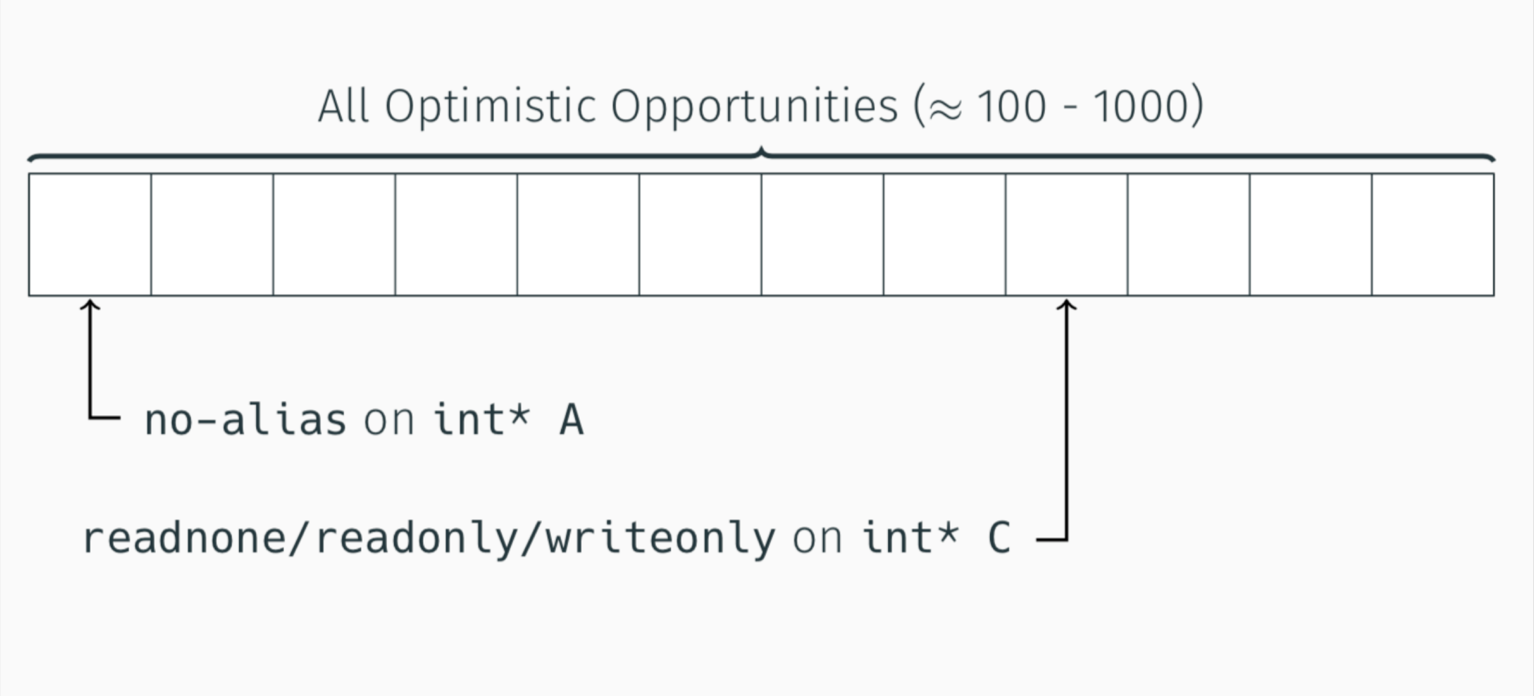
IN A NUTSHELL

```
void baz(__attribute__((readonly)) int *A);
```

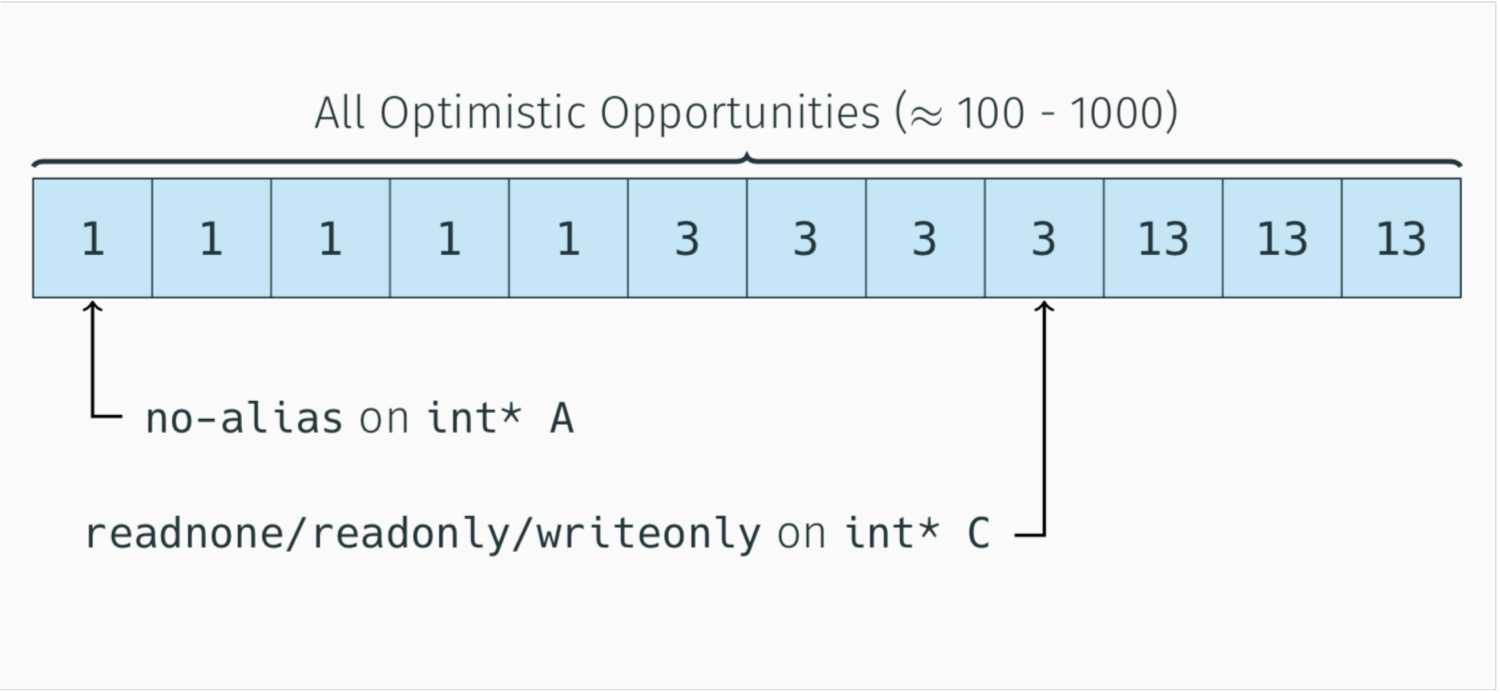
>> clang -O3 ...

>> verify.sh --> Success

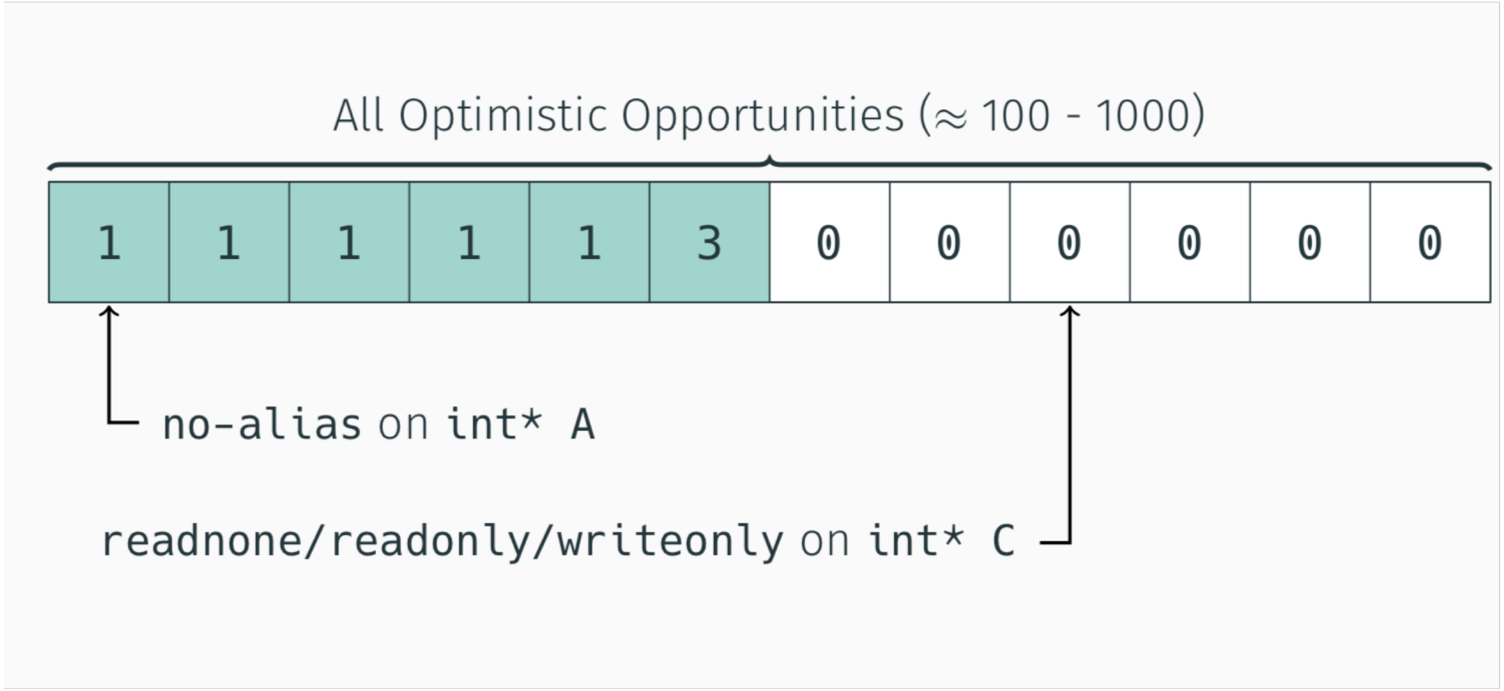
OPTIMISTIC OPPORTUNITIES



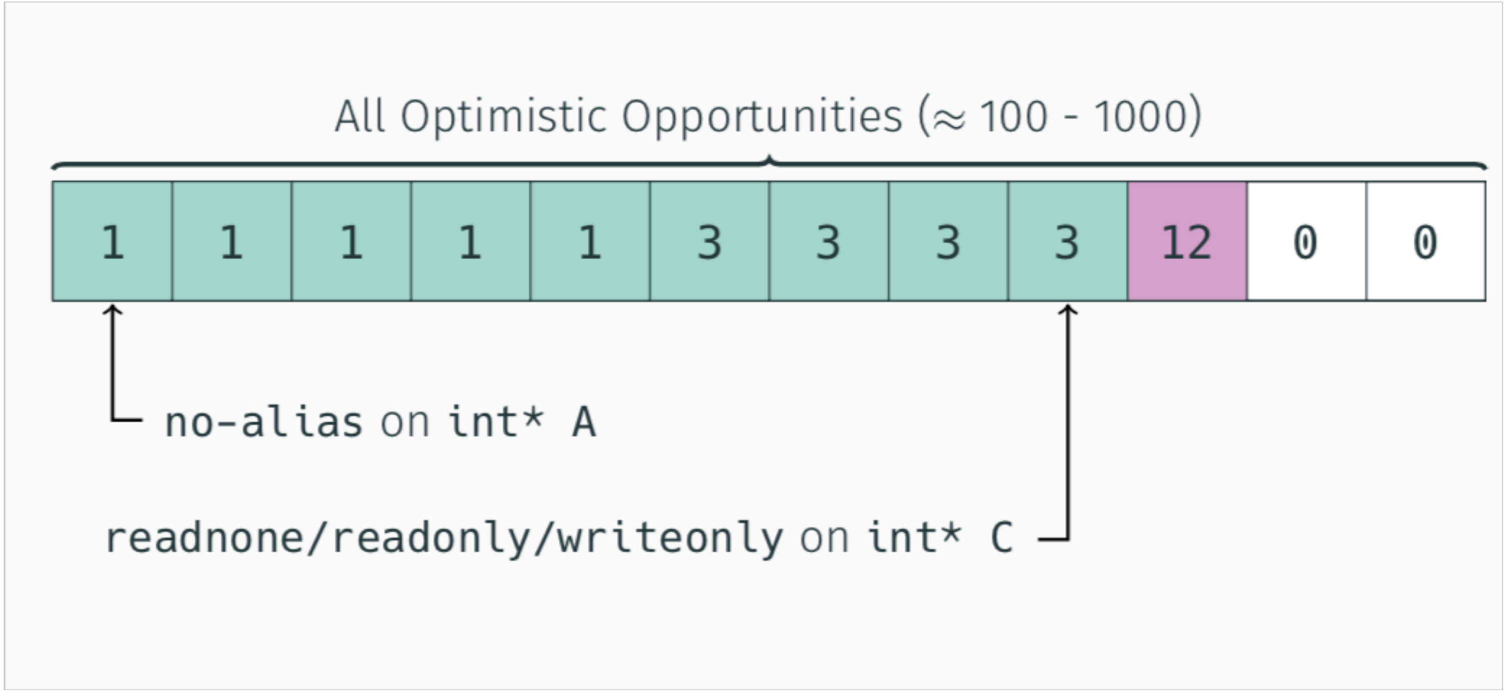
MARK THEM ALL OPTIMISTIC



SEARCH FOR VALID

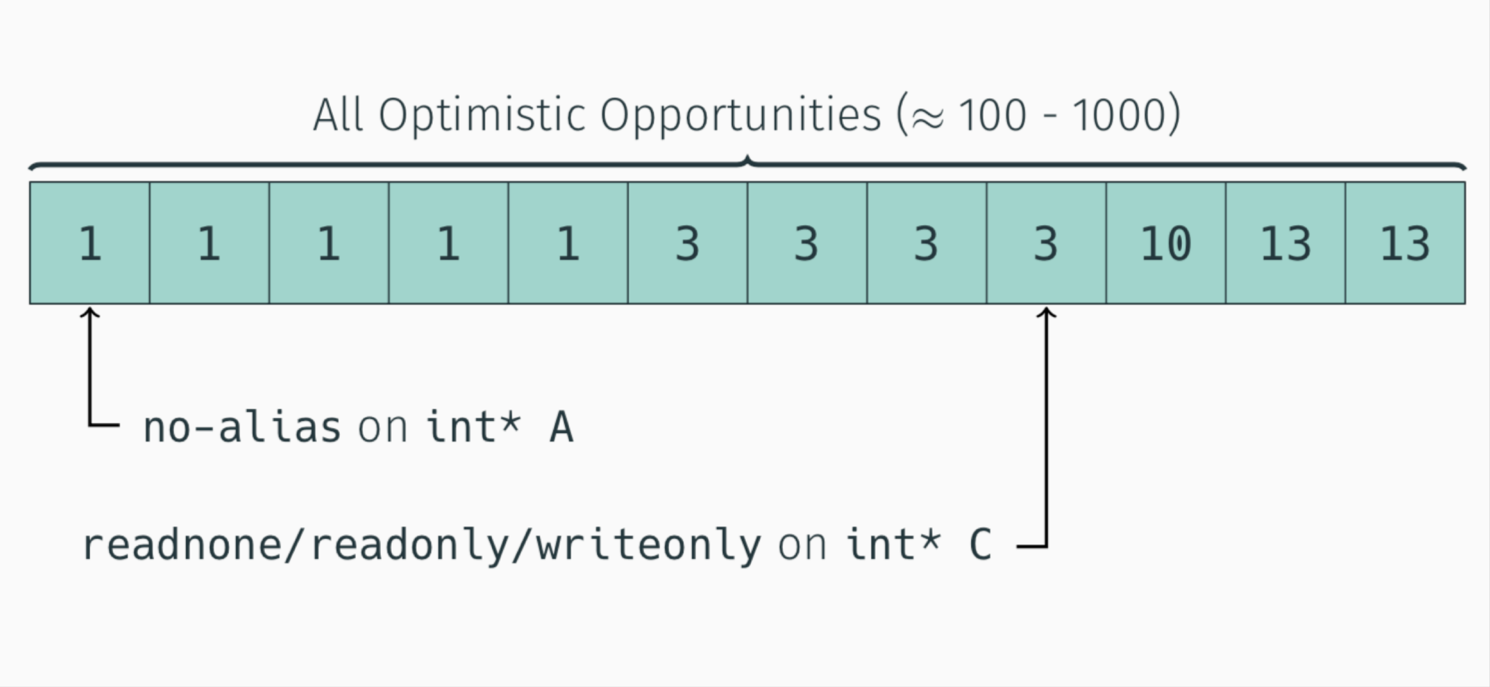


SEARCH



All Optimistic Opportunities (≈ 100 - 1000)
1
1
1
1
1
3
3
3
3
12
0
0
no-alias on int* A
readnone/readonly/writeonly on int* C

OPTIMISTIC CHOICES



OPPORTUNITY EXAMPLE – FUNCTION SIDE-EFFECTS

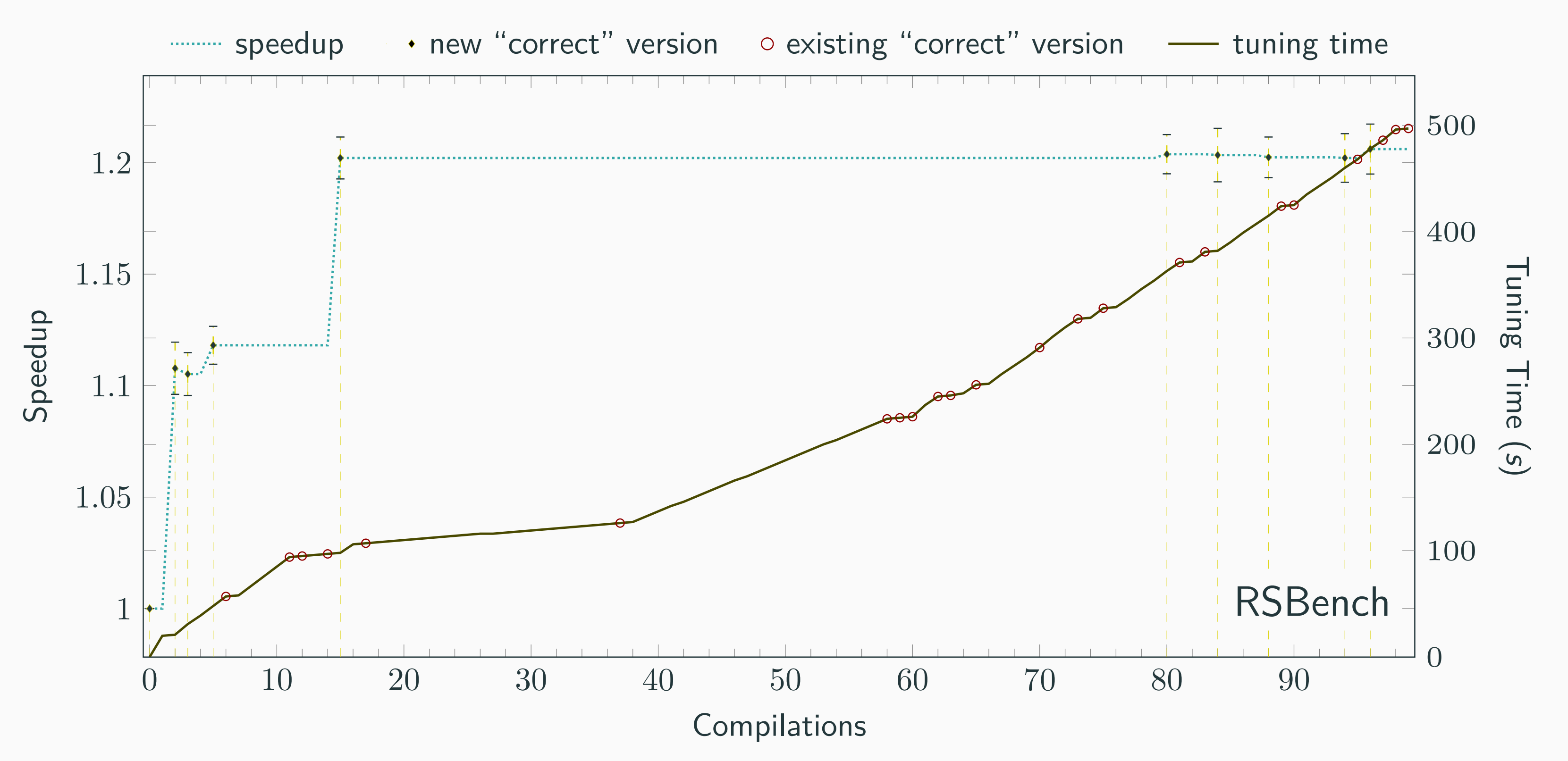
13. **speculatable** (and **readnone**)
12. **readnone**
11. **readonly** and **inaccessiblememonly**
10. **readonly** and **argmemonly**
9. **readonly** and **inaccessiblemem_or_argmemonly**
8. **readonly**
7. **writeonly** and **inaccessiblememonly**
6. **writeonly** and **argmemonly**
5. **writeonly** and **inaccessiblemem_or_argmemonly**
4. **writeonly**
3. **inaccessiblememonly**
2. **argmemonly**
1. **inaccessiblemem_or_argmemonly**
0. no annotation, original code

ANNOTATION OPPORTUNITIES

- Potentially aliasing pointers
- Potentially escaping pointers
- Potentially overflowing computations
- Potential runtime exceptions in functions
- Potentially parallel loops
- Externally visible functions
- Potentially non-dereferenceable pointers
- Unknown pointer alignment
- Unknown control flow choices
- Potentially invariant memory locations
- Unknown function return values
- Unknown pointer usage
- Potential undefined behavior in functions
- Unknown function side-effects

OPTIMISTIC TUNER RESULTS

Proxy Application	Problem Size / Run Configuration	# Successful Compilations	# New Versions	Optimistic Opportunities Taken
RSBench	-p 300000	32	9 (28.1%)	225/240 (93.8%)
XSBench	-p 500000	47	5 (10.6%)	129/141 (91.5%)
PathFinder	-x 4kx750.adj_list	62	22 (35.5%)	264/299 (88.3%)
CoMD	-x 40 –y 40 –z 40	49	13 (26.5%)	179/194 (92.3%)
Pennant	leblancbig.pnt	69	12 (17.4%)	610/689 (88.5%)
MiniGMG	6 2 2 2 1 1 1	16	4 (25.0%)	479/479 (100%)

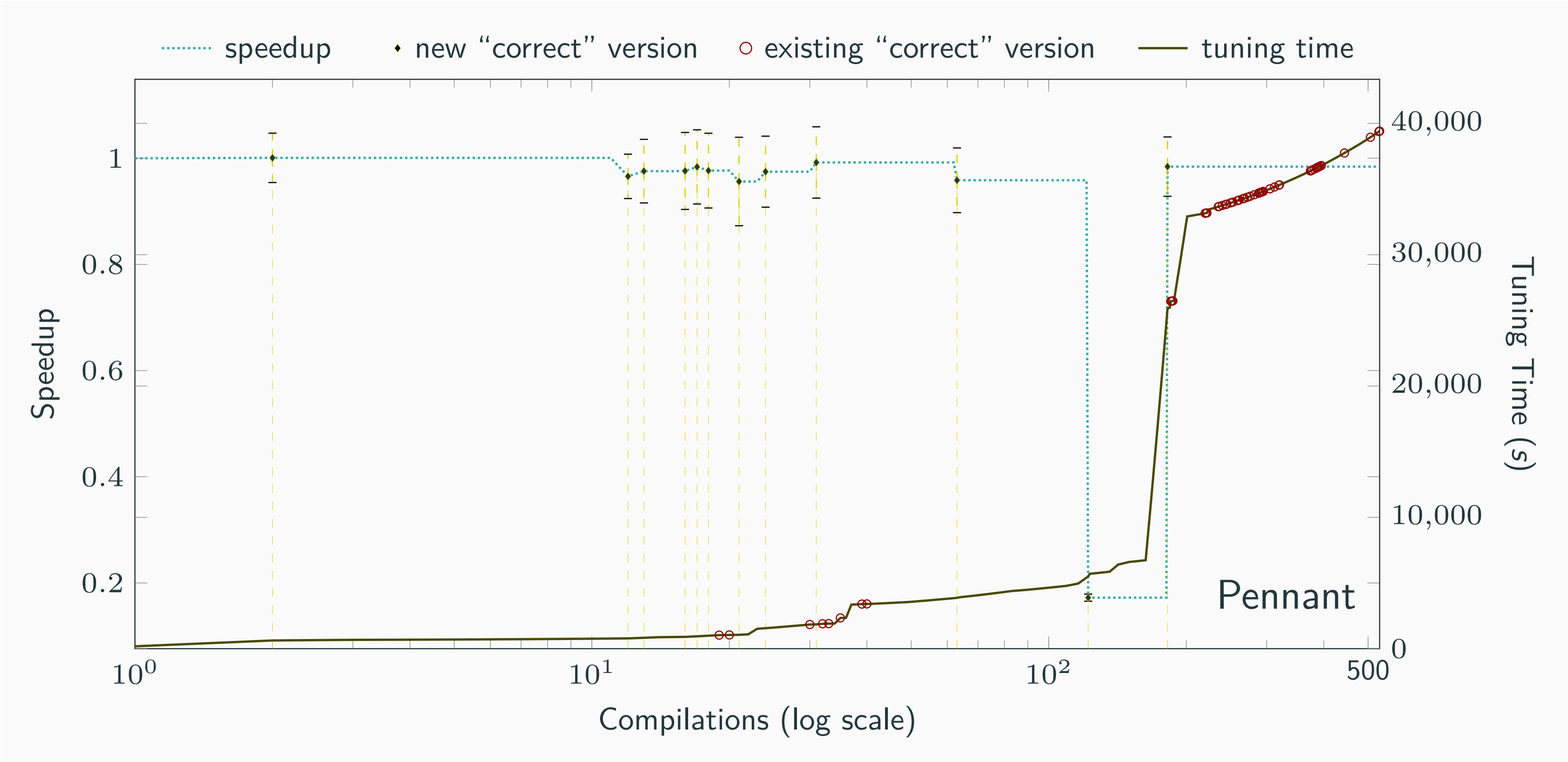


speedup
new “correct” version
existing “correct” version
tuning time
Speedup
1.2
1.15
1.1
1.05
1
Tuning Time (s)
500
400
300
200
100
0
RSBench
0
10
20
30
40
50
60
70
80
90
Compilations

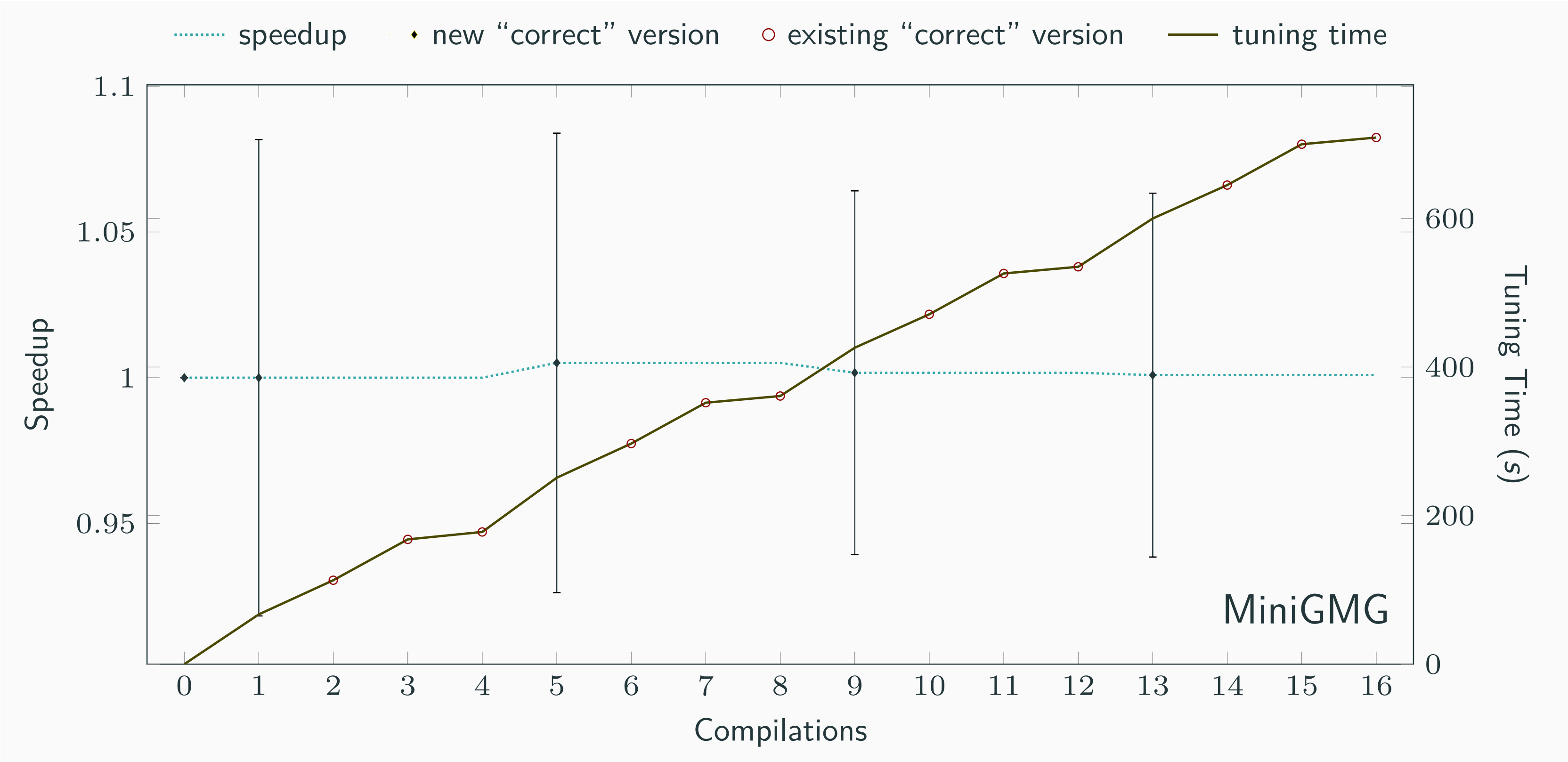

speedup
new “correct” version
existing “correct” version
tuning time
Speedup
Tuning Time (s)
Compilations
XSBench
1.2
1.15
1.1
1.05
1
400
300
200
100
0
10
20
30
40
50
60
70
80
90

speedup
new “correct” version
existing “correct” version
tuning time
Speedup
1.2
1.15
1.1
1.05
1
0.95
Tuning Time (s)
4,000
3,000
2,000
1,000
0
0
20
40
60
80
100
120
140
160
180
200
220
240
Compilations
PathFinder


speedup
new "correct" version
existing "correct" version
tuning time
Speedup
Tuning Time (s)
Compilations
CoMD
1.06
1.04
1.02
1
0.98
0.96
0.94
2,500
2,000
1,500
1,000
500
0
0
10
20
30
40
50
60
70
80
90
100
110
120



speedup
new “correct” version
existing “correct” version
tuning time
Speedup
1
0.8
0.6
0.4
0.2
Tuning Time (s)
40,000
30,000
20,000
10,000
0
Compilations (log scale)
10^0
10^1
10^2
500
Pennant


speedup
new "correct" version
existing "correct" version
tuning time
Speedup
1.1
1.05
1
0.95
Tuning Time (s)
600
400
200
0
MiniGMG
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
Compilations

COMPARISON TO LTO

Performance Gap with LTO as Baseline

Proxy Application	LTO	thin-LTO
RSBench	2.86%	5.68%
XSBench	14.03%	41.23%
PathFinder	3.67%	4.79%
CoMD	4.75%	4.48%
Pennant	-1.13%	-1.14%
MiniGMG	0.73%	0.79%

OPTIMISTIC SUGGESTIONS

OPTIMISTIC OPPORTUNITIES WITH CHOICES MADE

RSBench

1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1	1	1	1	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2
2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	2	1	1	1	1	1	1	1	1	1	0	0	0
1	1	1	1	1	1	1	1	1	3	3	3	3	3	3	3	3	3	3	3	1	3	1	3	0	7	0	7	7				

PERFORMANCE CRITICAL OPTIMISTIC CHOICES

RSBench

0	0	0	0	0	1	0	0	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	3	0	0	0	0	0				

SUGGESTION EXAMPLES

```
xs_kernel.c:6:1: remark: internalize the function,
                 e.g., through 'static' or 'namespace { ... }'.
double complex fast_nuclear_W(double complex Z) {
^

In file included from xs_kernel.c:1:
rsbench.h:94:16: remark: provide better information on function memory
                 effects, e.g., through '__attribute__((pure))' or
                 '__attribute__((const))'
complex double fast_cexp( double complex z );
```

FUTURE WORK

- Improvements to the tool (suggestions and search)
- Additional results
- Identify information that causes regressions

- Understand if information was not useful or not used
- Collect statistics on addition information that does/does not change the binary

ACKNOWLEDGEMENTS

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of two U.S. Department of Energy organizations (Office of Science and the National Nuclear Security Administration) responsible for the planning and preparation of a capable exascale ecosystem, including software, applications, hardware, advanced system engineering, and early testbed platforms, in support of the nation's exascale computing imperative.

THANK YOU