

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

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



Hewlett Packard
Enterprise



Lawrence
Livermore
National
Laboratory

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>

The left screenshot shows the 'ECP Proxy Apps Suite' page for 'Release 1.0', dated October 31, 2017. It features a table of proxy applications:

Proxy App	Version	Website	GitHub
AMG	1.0	Website	GitHub
ExaMiniMD	1.0	Website	GitHub
Laghos	1.0	Website	GitHub
miniAMR	1.4.0	Website	GitHub
miniFE	2.1.0	Website	GitHub
miniTri	1.0	Website	GitHub
NEKbone	17.0	Website	GitHub
SHaite	1.0	Website	GitHub
SWFFT	1.0	Website	GitHub
XSBench	14	Website	GitHub

The right screenshot shows the 'Catalog' page, which displays a grid of application cards. Each card includes the application name, supported programming models, a brief description, and icons for documentation, GitHub, and releases.

Application	Supported Languages	Description
AMG C	C	AMG is a parallel algebraic multigrid solver for linear systems arising from problems on unstructured grids.
ASPA C++	C++	ASPA Proxy Application, Multi-scale, adaptive sampling, materials science proxy.
CLAMR C++	C++	CLAMR is a cell-based adaptive mesh refinement mini-app developed as a testbed for hybrid algorithm development using MPI and OpenCL.
CloverLeaf3D Fortran	Fortran	3D version of a minapp that solves compressible Euler equations on a Cartesian grid.
CloverLeaf Fortran	Fortran	A minapp that solves the compressible Euler equations on a Cartesian grid.
CoGL C++	C++	Analyzes pattern formation in ferroelectric materials and tests in situ visualization.
CoHMM C	C	A proxy application for the Heterogeneous Multiscale Method (HMM) augmented with adaptive sampling.
CoMD C	C	A classical molecular dynamics proxy application implemented in multiple programming models.
CoSP2 C	C	CoSP2 implements typical linear algebra algorithms and workflows for a quantum molecular dynamics (QMD) electronic structure code.
EBMS C	C	A minapp for the Energy Banding Monte Carlo (EBMC) neutron transport simulation code.
Ember Communication Patterns C and MPI/SHMEM	C, MPI, SHMEM	Ember code components represent highly.
ExaMiniMD C++	C++	A proxy application and research vehicle for.

ECP PROXY APPLICATION PROJECT

ECP Proxy Applications

Home

ECP Proxy Apps Suite

Catalog

Submit App

Standards

How-Tos

Reports

Team

The online collection for exascale applications

A major goal of the Exascale Proxy Applications Project is to improve the quality of proxies created by ECP and maximize the benefit received from their use. To accomplish this goal, an ECP proxy app suite composed of proxies developed by ECP projects that represent the most important features (especially performance) of exascale applications will be created.

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?

Intel Software Development Emulator

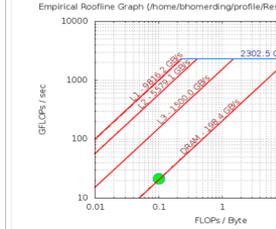
Intel SDE	HPCCG
Arithmetic intensity	0.103
FLOPS per Inst	0.442
FLOPS per FP Inst	1.71
Bytes per Load Inst	7.96
Bytes per Store Inst	7.52

Roofline - Intel(R) Xeon(R) Platinum 8180M CPU

112 Threads - 56 Cores 3200.0 Mhz

Op/Sec	L1 B/W	L2 B/W	L3 B/W	DRAM B/W
1 Thread	159.33	91.42	47.08	21.27
56 Threads	9816.2	5579.1	1050.00*	198.4
112 Threads	9912.56	5573.58	1050.00*	203.13

*L3 BW ERT unable to recognize. Very short plateau (estimate)



@ Note: DRAM BW bound?

HPCCG_sparvem.cpp

Threads (Time)	IPC per Core	Loads per Cycle	L1 Hits per Cycle	L1 Miss Ratio	L2 Miss Ratio	L3 Miss Ratio
1 (83.1%)	1.39	0.77	0.68	1.86%	42.24%	72.59%
56 (70.4%)	0.47	0.25	0.22	1.74%	42.51%	68.42%
112 (65.5%)	0.48	0.13	0.12	2.19%	42.56%	75.15%

```

66 int HPC_sparsem( HPC_Sparse_Matrix *A,
67                 const double *const x, double *const y)
68 {
69
70     const int nrow = (const int) A->nrow;
71
72     #ifdef USING_OMP
73     #pragma omp parallel for
74     #endif
75     for (int i=0; i<nrow; i++)
76     {
77         double sum = 0.0;
78         const double *const cur_vals =
79             (const double *) A->ptr_to_vals_in_row[i];
80
81         const int *const cur_inds =
82             (const int *) A->ptr_to_inds_in_row[i];
83
84         const int cur_nnz = (const int) A->nnz_in_row[i];
85
86         for (int j=0; j<cur_nnz; j++)
87             sum = cur_vals[j]+cur_inds[j];
88         y[i] = sum;
89     }
90     return(0);
91 }
    
```

Threads (Time)	IPC per Core	Loads per Cycle	L1 Hits per Cycle	L1 Miss Ratio	L2 Miss Ratio	L3 Miss Ratio
1 (64.5%)	1.32	0.62	0.55	1.92%	43.27%	71.10%
56 (63.7%)	0.41	0.20	0.17	1.78%	44.29%	63.23%
112 (60.1%)	0.47	0.13	0.11	2.20%	43.91%	74.23%

```

86     for (int j=0; j<cur_nnz; j++)
87         sum = cur_vals[j]+cur_inds[j];
88     y[i] = sum;
89 }
90 return(0);
91 }
    
```

HPCCG

A simple conjugate gradient benchmark code for a 3D chimney domain on an arbitrary number of processors.

Parameters

```

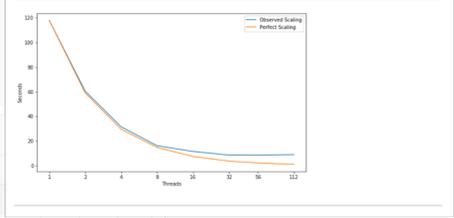
Compiler = gcc (GCC) 10.4.1 20170818
Build_Flags = -g -O3 -march=native -fsee-vectorize -qopenmp -DUSING_OMP
Run_Parameters = 256 256 256
    
```

Scaling

```

import matplotlib
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
matplotlib.in_line
plt.rcParams["figure.figsize"] = (10,6)

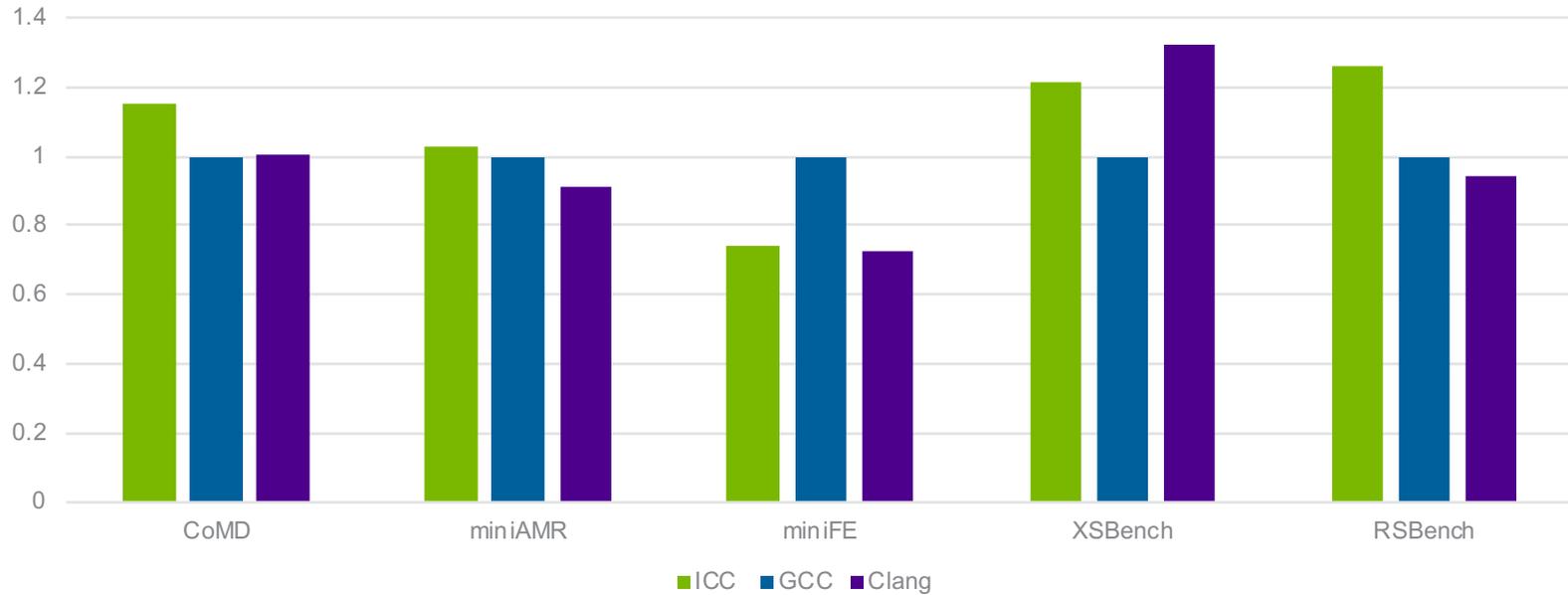
threads = [1, 2, 4, 8, 16, 32, 56, 112]
observedScaling = [118.0, 58.5, 31.5, 16.125, 8.5625, 8.482142857142858, 8.928571428571429]
perfectScaling = [118.0, 59.0, 29.5, 14.75, 7.375, 3.5875, 2.187142857142857, 1.9035714285714286]
fig, ax = plt.subplots(); ax.set_xlabel("Threads"); ax.set_ylabel("GFLOPs")
ax.set_xaxis(1); ax.set_major_formatter(matplotlib.ticker.ScalarFormatter())
ax.plot(threads, observedScaling, label="Observed Scaling")
ax.plot(threads, perfectScaling, label="Perfect Scaling")
ax.legend(); ax.set_xlabel("Threads"); ax.set_ylabel("Seconds")
plt.show()
    
```



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



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
```

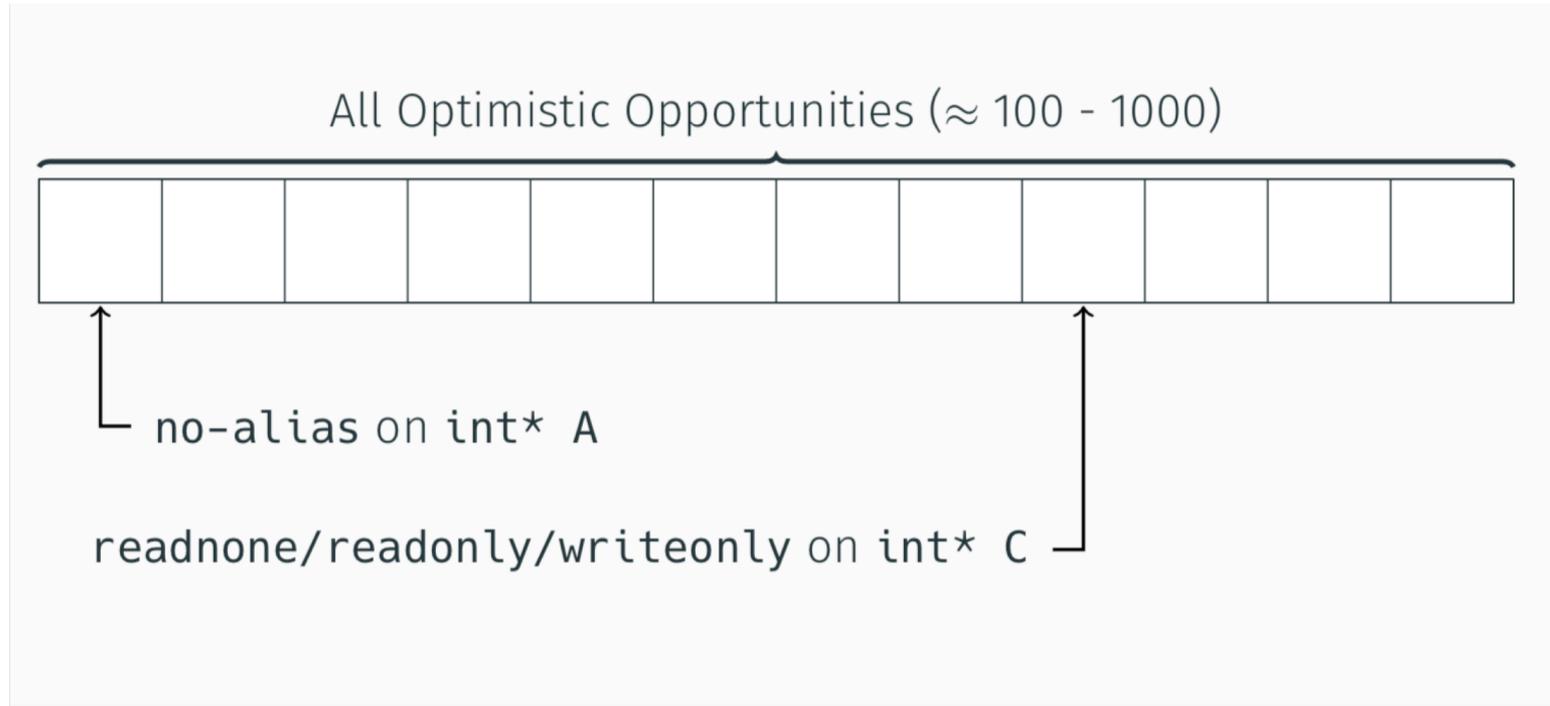
IN A NUTSHELL

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

```
>> clang -O3 ...
```

```
>> verify.sh --> Success
```

OPTIMISTIC OPPORTUNITIES



MARK THEM ALL OPTIMISTIC

All Optimistic Opportunities ($\approx 100 - 1000$)



no-alias on `int* A`

readnone/readonly/writeonly on `int* C`

SEARCH FOR VALID

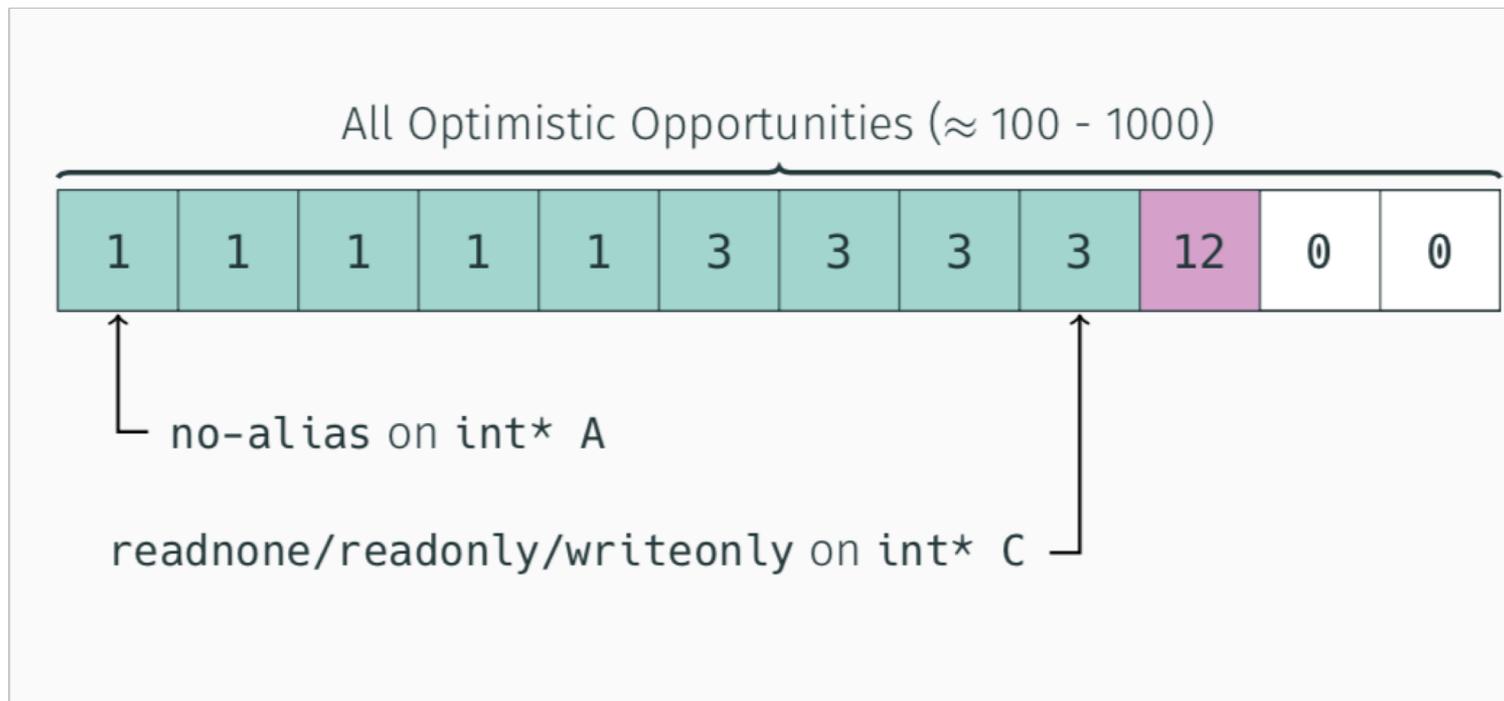
All Optimistic Opportunities ($\approx 100 - 1000$)



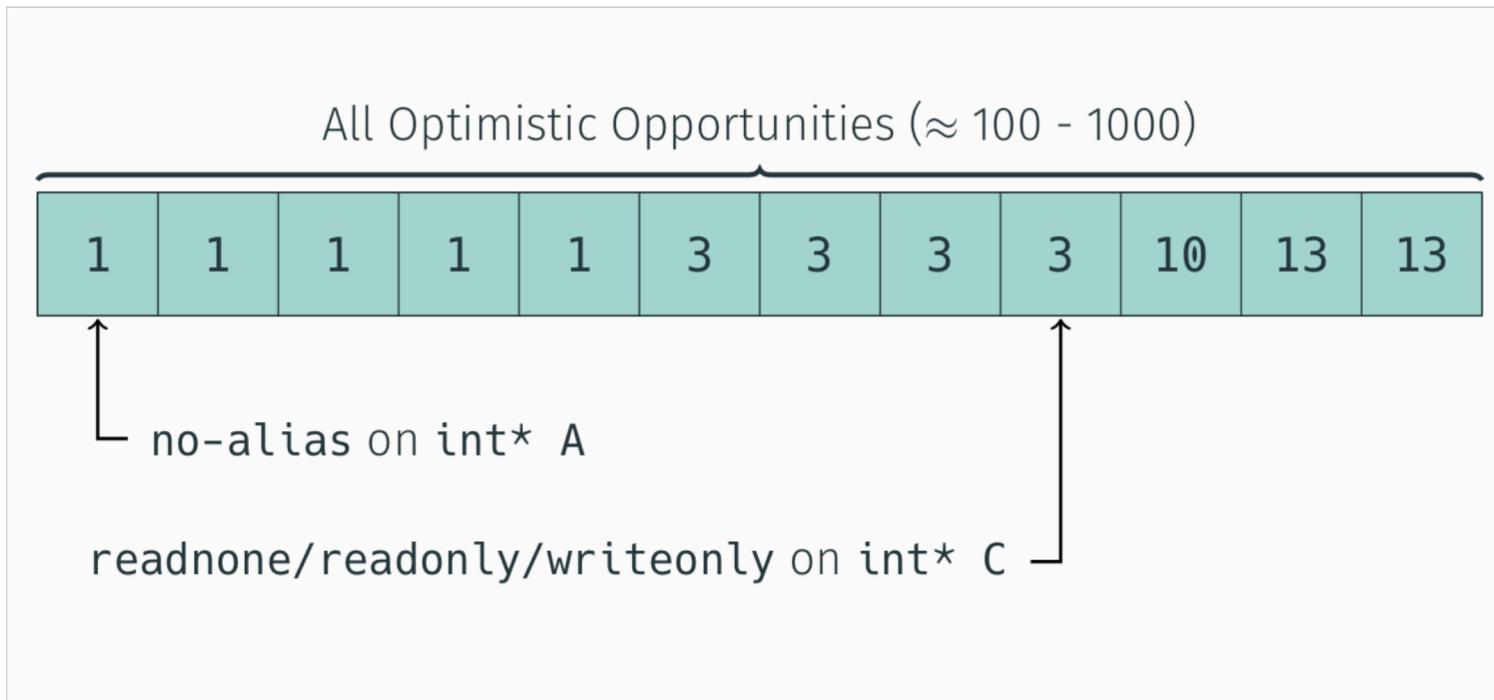
no-alias on `int* A`

readnone/readonly/writeln on `int* C`

SEARCH



OPTIMISTIC CHOICES



OPPORTUNITY EXAMPLE – FUNCTION SIDE-EFFECTS

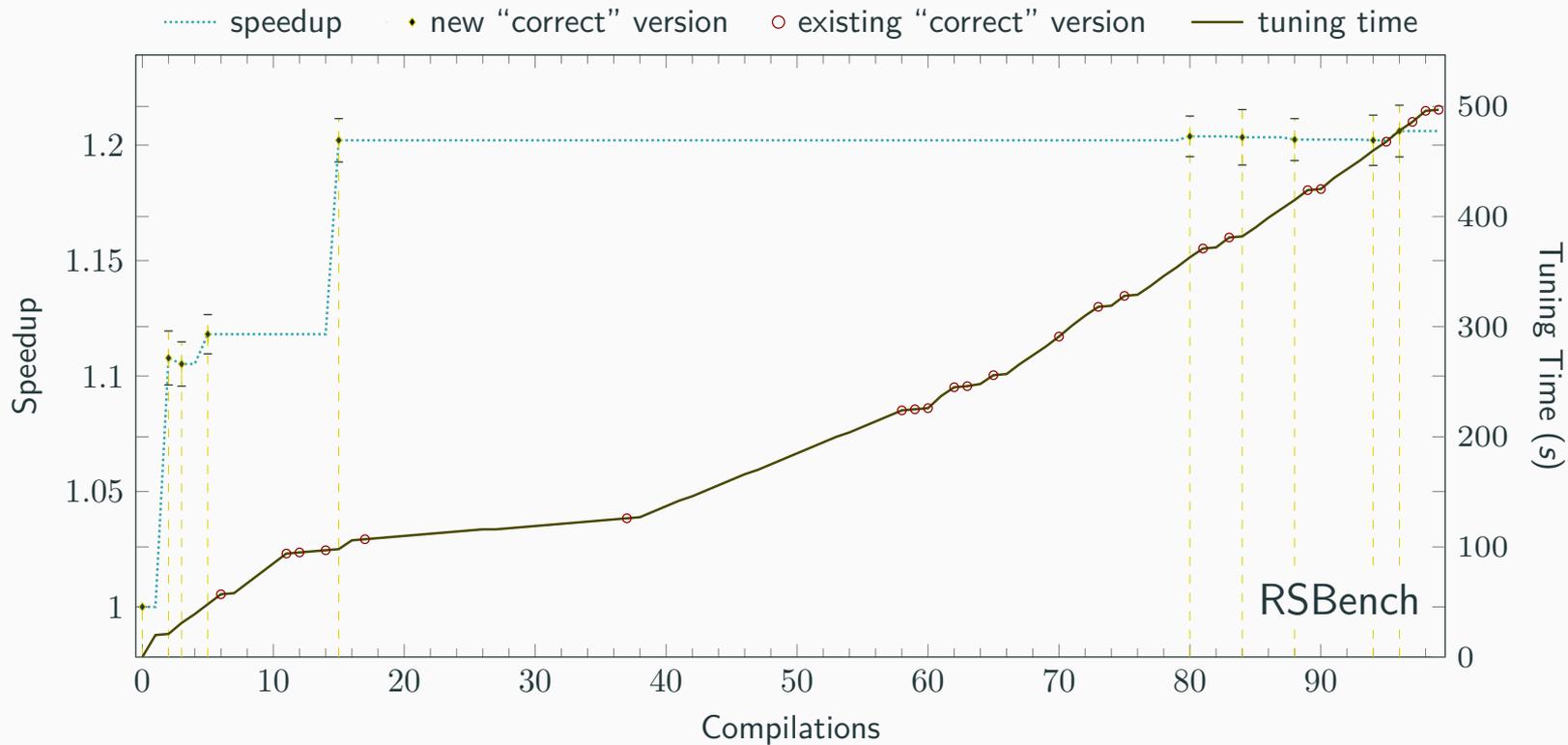
13. **speculatable** (and **readnone**)
12. **readnone**
11. **readonly** and **inaccessiblememonly**
10. **readonly** and **argmemonly**
9. **readonly** and **inaccessiblemem_or_argmemonly**
8. **readonly**
7. **writeonly** and **inaccessiblememonly**
6. **writeonly** and **argmemonly**
5. **writeonly**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

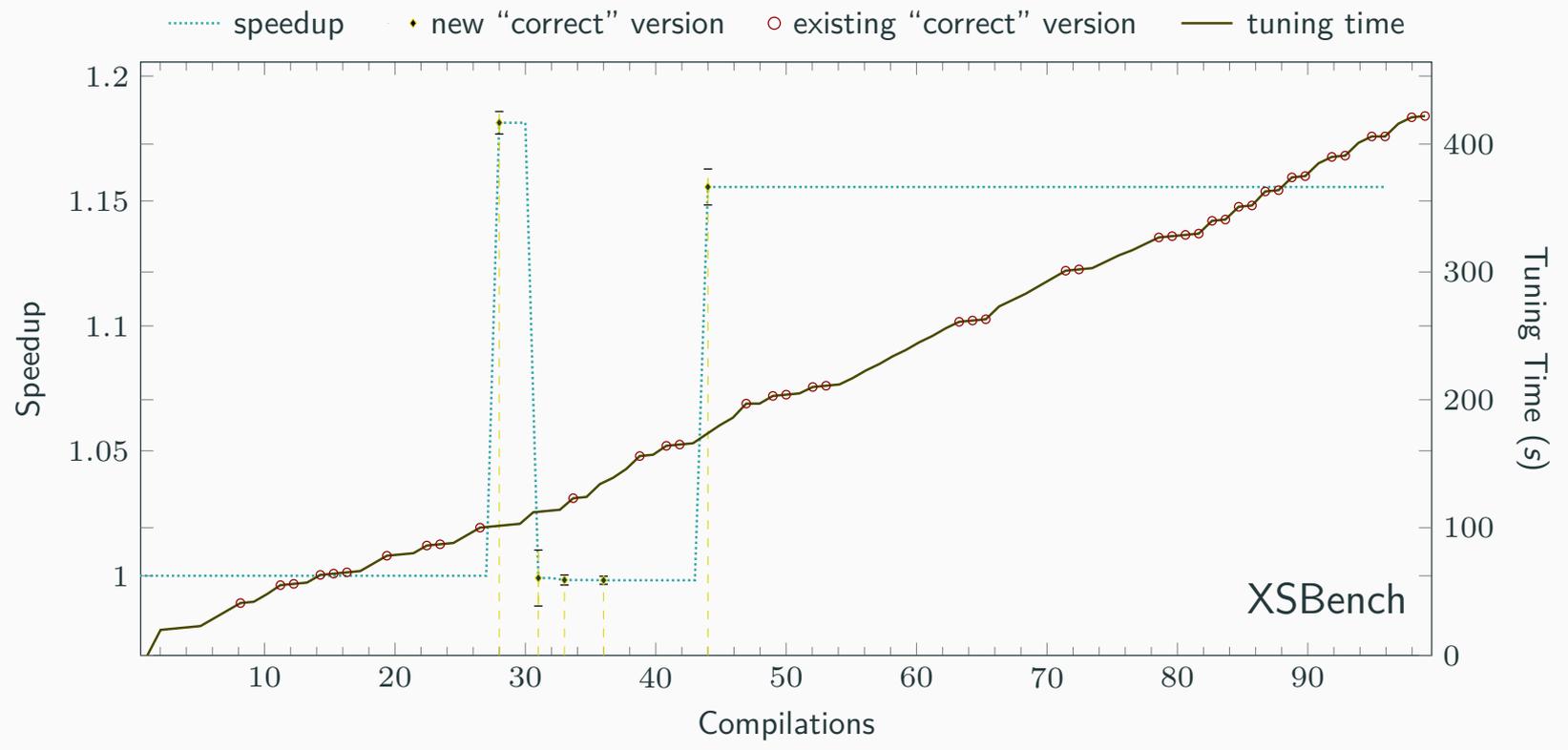
ANNOTATION OPPORTUNITIES

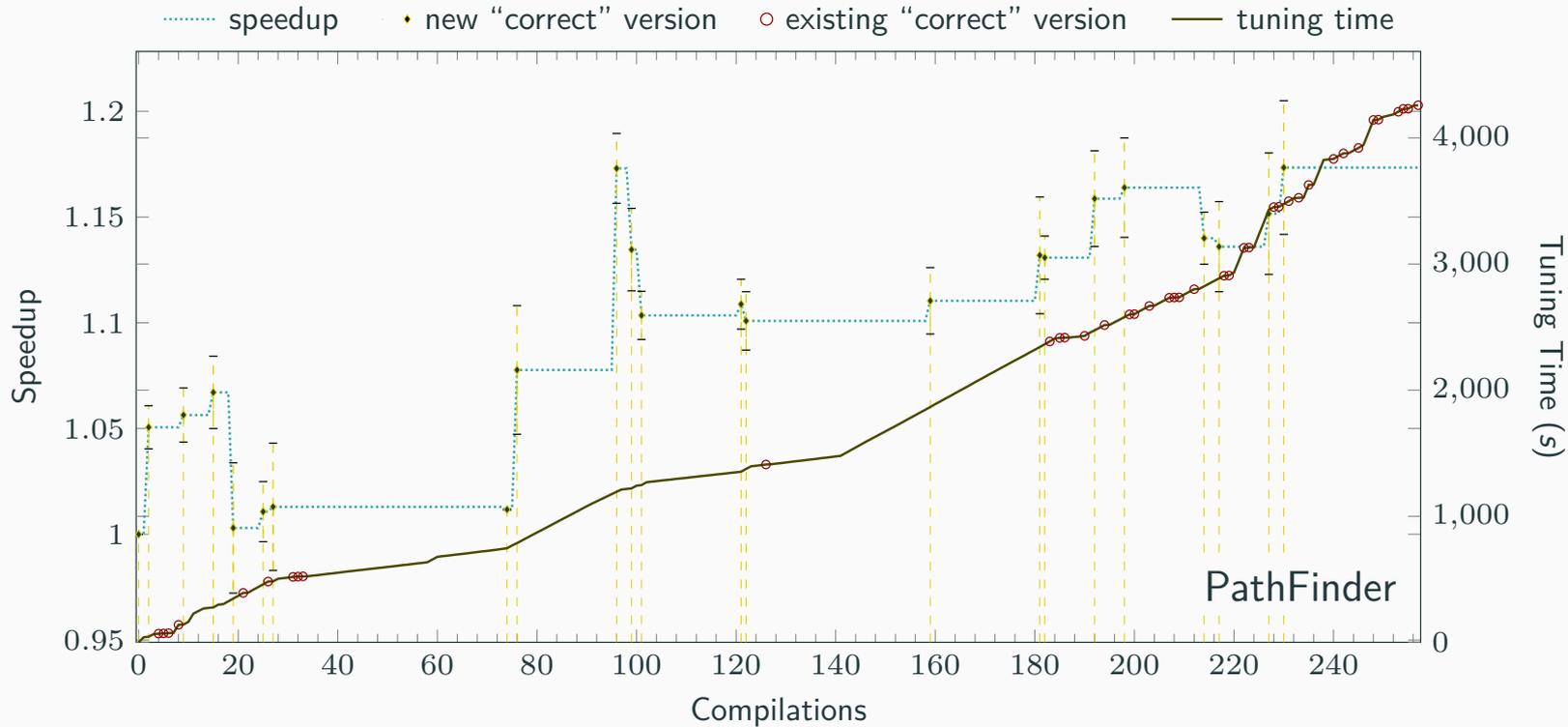
- Potentially aliasing pointers
- Potentially escaping pointers
- Potentially overflowing computations
- Potential runtime exceptions in functions
- Potentially parallel loops
- Externally visible functions
- Potentially non-dereferenceable pointers
- Unknown pointer alignment
- Unknown control flow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

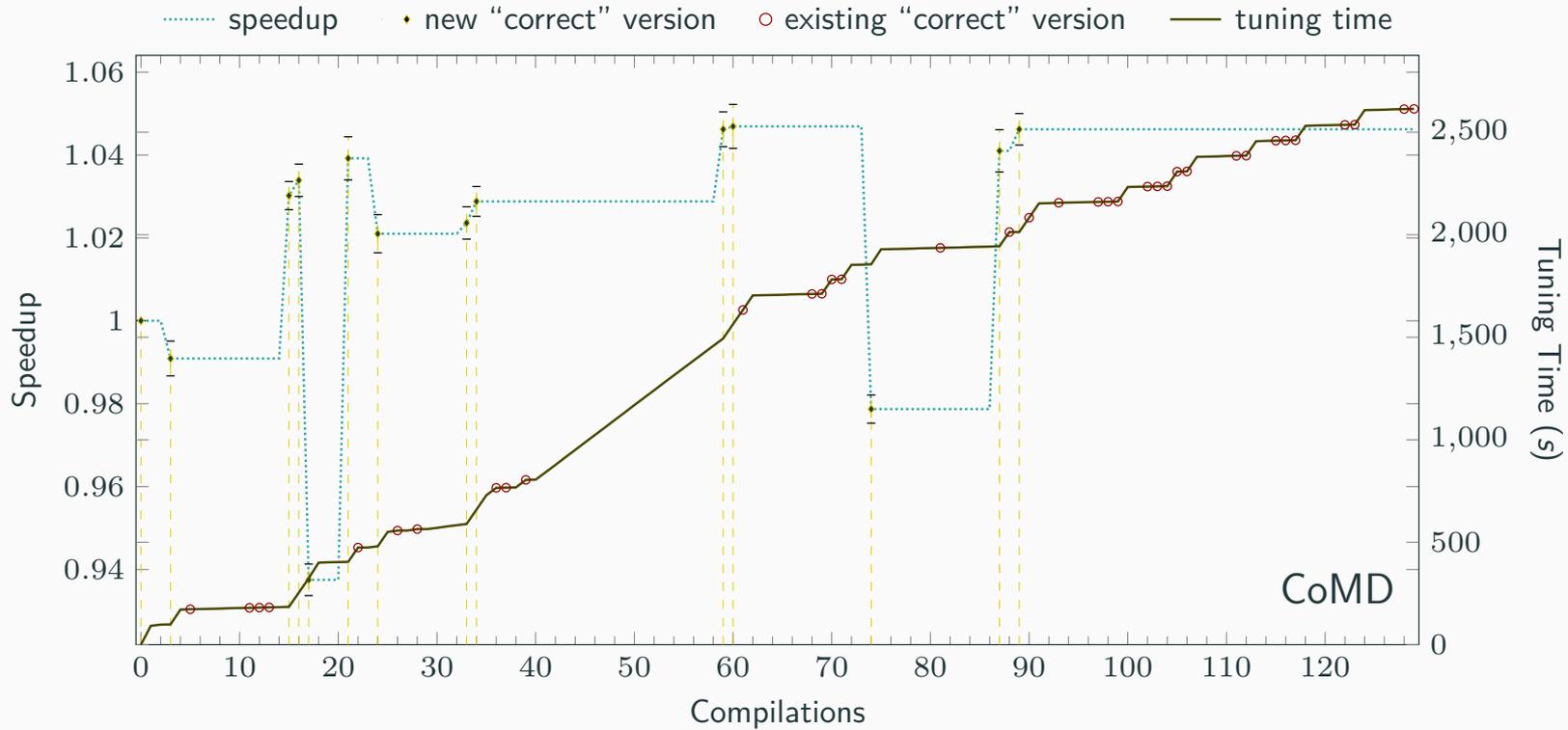
OPTIMISTIC TUNER RESULTS

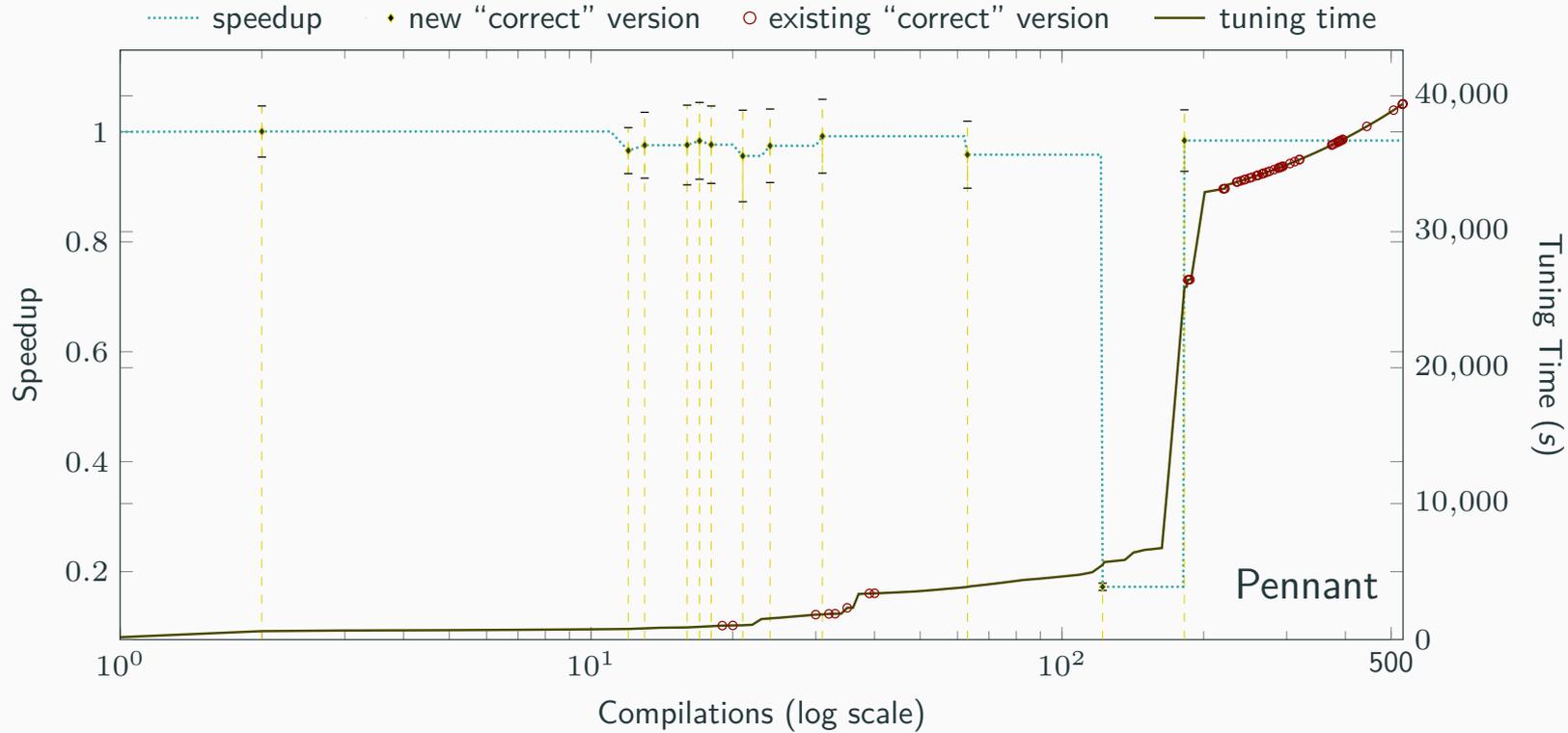
Proxy Application	Problem Size / Run Configuration	# Successful Compilations	# New Versions	Optimistic Opportunities Taken
RSBench	-p 300000	32	9 (28.1%)	225/240 (93.8%)
XSBench	-p 500000	47	5 (10.6%)	129/141 (91.5%)
PathFinder	-x 4kx750.adj_list	62	22 (35.5%)	264/299 (88.3%)
CoMD	-x 40 -y 40 -z 40	49	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%)













MiniGMG

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



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

