Performance Exploration Through Optimistic Static Program Annotations

ISC'19 — June 18, 2019 — Frankfurt, Germany

Johannes Doerfert, Brian Homerding, and Hal Finkel

Leadership Computing Facility
Argonne National Laboratory
https://www.alcf.anl.gov/

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.

Motivation

```
double foo(int N, double A[N]) {
  double v = 0.0;
  for (int i = 0; i < N; i++)
    v = v + A[i];
  return v;
}
```

```
>> clang -O3

double foo(int N, double A[N]) {
  double v = 0.0;
  for (int i = 0; i < N; i++)
    v = v + A[i];
  return v;
}
```

```
>> clang -O3

  double foo(int N, double A[N]) {
    double v = 0.0;
    for (int i = 0; i < N; i++)
      v =
    retur
  }
```

Was it (properly) optimized?

Is this as good as it gets?

```
>> clang -O3

double foo(int N, double A[N]) {
  double v = 0.0;
  for (int i = 0; i < N; i++)
    v =
  retur
}
```

Was it (properly) optimized?

Is this as good as it gets?

Let's ask the compiler, it will know!

```
>> clang -O3 --Rpass-analysis=loop-vectorize

   double foo(int N, double A[N]) {
     double v = 0.0;
     for (int i = 0; i < N; i++)
       v = v + A[i];
     return v;
   }
```

```
>> clang -O3 --Rpass-analysis=loop-vectorize
```

```c
double foo(int N, double A[N]) {
  double v = 0.0;
  for (int i = 0; i < N; i++)
    v = v + A[i];
  return v;
}
```

```
remark: loop not vectorized: cannot prove it is safe to
reorder floating-point operations; allow reordering by
specifying '#pragma clang loop vectorize(enable)' before
the loop or by providing the compiler option '-ffast-math'.
```

```
>> clang
    doubl
      dou
      for
        v
      ret
    }
    remar
    reord
    spec
    the                                                    h'.
```

Problems include:

- **Too many** transformations to manually digest and act on **remarks**.

```
>> clang
   doubl
     dou
     for
       v
     ret
   }
   remar
   reorc
   spec
   the                                                          h'.
```

Problems include:

- Too many transformations to manually digest and act on remarks.
- **Not all** transformations issue (precise) remarks **with suggestions**.

```
>> clang
  doubl
    dou
    for
      v
    ret
  }
  rema
  reor
  spec
  the                                   h'.
```

Problems include:

- Too many transformations to manually digest and act on remarks.
- Not all transformations issue (precise) remarks with suggestions.
- **Manually** following suggestions is **tedious** and may **not** yield **performance**.

```
>> clang
   doubl
     dou
     for
       v
     ret
   }
   remar
   reord
   spec
   the                                                                  h'.
```

Problems include:

- Too many transformations to manually digest and act on remarks.
- Not all transformations issue (precise) remarks with suggestions.
- Manually following suggestions is tedious and may not yield performance.
- **Complicated to "try"** coarse-grained **options** like `-ffast-math` structured and targeted.

```
>> clang
    doubl
      dou
      for
        v
      ret
    }
    remar
    reorc
    spec
    the                                                           h'.
```

Problems include:

- Too many transformations to manually digest and act on remarks.
- Not all transformations issue (precise) remarks with suggestions.
- Manually following suggestions is tedious and may not yield performance.
- Complicated to "try" coarse-grained options like `-ffast-math` structured and targeted.
- ...

>> 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;
}
```

```
>> clang -O3

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

int bar
  int s
  std::
  external(&sum, locP);

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

Was it (properly) optimized?

Is this as good as it gets?

```
>> clang -O3

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

int bar
  int s
  std::
  exter

  for (
    sum
  return sum;
}
```

Was it (properly) optimized?

Is this as good as it gets?

What should we ask the compiler? Is vectorization again the goal?

```
>> 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;
}
```

```
>> 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);
}
```

```
>> 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;
}
```

Optimistic Program Annotations

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

Optimistic Annotation in a Nutshell

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

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

Optimistic Annotation in a Nutshell

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Optimistic Annotation in a Nutshell — Non-Boolean Choices

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

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

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

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

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

```
void baz(__attribute__((align_value(64)))
         __attribute__((readonly)) int *A);
```

```
void baz(__attribute__((align_value(64)))
         __attribute__((readonly)) int *A);
```

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

```
void baz(__attribute__((align_value(64)))
         __attribute__((readonly)) int *A);
```

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

```
    void baz(__attribute__((nonnull))
             __attribute__((align_value(64)))
             __attribute__((readonly)) int *A);
>> clang -O3 ...
```

Annotation Opportunities

- potentially aliasing pointers (`__restrict__`)

Annotation Opportunities

- potentially aliasing pointers (`__restrict__`)
- potentially escaping pointers

Annotation Opportunities

- potentially aliasing pointers (`__restrict__`)
- potentially escaping pointers
- potentially overflowing computations

Annotation Opportunities

- potentially aliasing pointers (`__restrict__`)
- 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

13. **`speculatable`** (and **`readnone`**)

13. **`speculatable`** (and **`readnone`**)
12. **`readnone`**

Opportunity Example — Function Side-Effects

13. **speculatable** (and **readnone**)
12. **readnone**
11. **readonly** and **inaccessiblememonly**

13. **`speculatable`** (and **`readnone`**)
12. **`readnone`**
11. **`readonly`** and **`inaccessiblememonly`**
10. **`readonly`** and **`argmemonly`**

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

Implementation

Program Code

Tuning Config

Test Suite

Program Code

⇓

Augmented Compiler

Tuning Config

⇓

Driver

Test Suite

Program Code

Tuning Config

Augmented Compiler

Driver

Test Suite

Program Code

Tuning Config

Augmented Compiler

Driver

Compiled Code

Test Suite

Program Code
Tuning Config
Augmented Compiler
Driver
Compiled Code
Test Suite

Program Code

Tuning Config

Augmented Compiler

Driver

Result

Compiled Code

Test Suite

```
{
    "source_files": [
        {
            "path": "main.c",
            "options": ["-ffast-math", "-O3"],
            "only_functions": ["foo", "bar"]
        }
    ],
    "executable": "./a.out",
    "make_cmd": "make",
    "verify_cmd": "verify.sh"
}
```

Configuration File — The User Provided Input

```json
{
    "source_files": [
        {
            "path": "main.c",
            "options": ["-ffast-math", "-O3"],
            "only_functions": ["foo", "bar"]
        }
    ],
    "executable": "./a.out",
    "make_cmd": "make",
    "verify_cmd": "verify.sh"
}
```

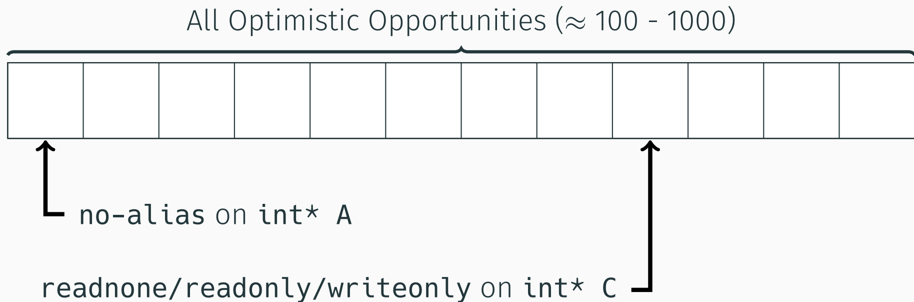
Configuration File — The User Provided Input

```
{
    "source_files": [
        {
            "path": "main.c",
            "options": ["-ffast-math", "-O3"],
            "only_functions": ["foo", "bar"]
        }
    ],
    "executable": "./a.out",
    "make_cmd": "make",
    "verify_cmd": "verify.sh"
}
```

Configuration File — The User Provided Input

```
{
    "source_files": [
        {
            "path": "main.c",
            "options": ["-ffast-math", "-O3"],
            "only_functions": ["foo", "bar"]
        }
    ],
    "executable": "./a.out",
    "make_cmd": "make",
    "verify_cmd": "verify.sh"
}
```

All Optimistic Opportunities (≈ 100 - 1000)


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

All Optimistic Opportunities (≈ 100 - 1000)

1	1	1	1	1	3	3	3	3	13	13	13

`no-alias` on `int* A`

`readnone/readonly/writeonly` on `int* C`

optimistic initialization

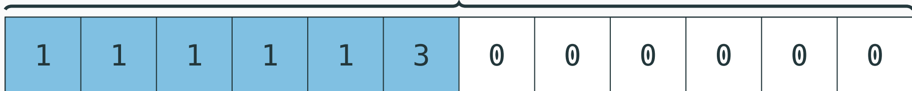
All Optimistic Opportunities (≈ 100 - 1000)

1	1	1	1	1	3	3	3	3	13	13	13

`no-alias` on `int* A`

`readnone/readonly/writeonly` on `int* C`

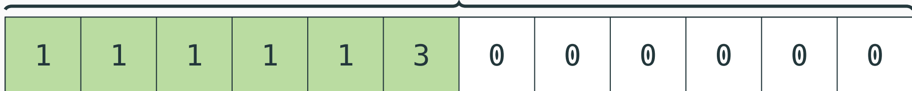
compile or verify failed



no-alias on `int* A`

readnone/readonly/writeonly on `int* C`

partial optimistic initialization



`no-alias` on `int* A`

`readnone/readonly/writeonly` on `int* C`

compile and verify succeeded

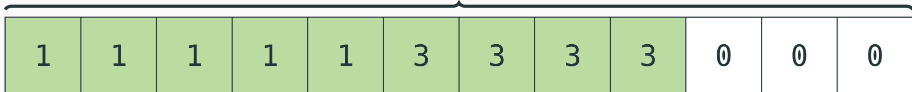
All Optimistic Opportunities (≈ 100 - 1000)

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

`no-alias` on `int* A`

`readnone/readonly/writeonly` on `int* C`

partial optimistic initialization



`no-alias` on `int* A`

`readnone/readonly/writeonly` on `int* C`

compile and verify succeeded

`no-alias` on `int* A`

`readnone/readonly/writeonly` on `int* C`

partial optimistic initialization

All Optimistic Opportunities (≈ 100 - 1000)

1	1	1	1	1	3	3	3	3	13	0	0

`no-alias` on `int* A`

`readnone/readonly/writeonly` on `int* C`

compile or verify failed

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`

reduce optimism

no-alias on int* A

readnone/readonly/writeonly on int* C

compile or verify failed

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

1	1	1	1	1	3	3	3	3	11	0	0

`no-alias` on `int* A`

`readnone/readonly/writeonly` on `int* C`

reduce optimism

All Optimistic Opportunities (≈ 100 - 1000)

1	1	1	1	1	3	3	3	3	11	0	0

`no-alias` on `int* A`

`readnone/readonly/writeonly` on `int* C`

compile or verify failed

All Optimistic Opportunities (≈ 100 - 1000)

1	1	1	1	1	3	3	3	3	10	0	0

`no-alias` on `int* A`

`readnone/readonly/writeonly` on `int* C`

reduce optimism

`no-alias` on `int* A`

`readnone/readonly/writeonly` on `int* C`

compile and verify succeeded

All Optimistic Opportunities (≈ 100 - 1000)

1	1	1	1	1	3	3	3	3	10	13	13

`no-alias` on `int* A`

`readnone/readonly/writeonly` on `int* C`

optimistic initialization

Compile Time Improvements – Effective Search

`no-alias` on `int* A`

`readnone/readonly/writeonly` on `int* C`

compile and verify succeeded

Evaluation

Evaluation — Benchmark Description

Benchmark	ID	# Threads	Base Time	# Successful Compilations	# New Versions
RSBench	(A)	72	8.56s		
XSBench	(B)	1	75.13s		
PathFinder	(C)	1	363.50s		
CoMD	(D)	72	44.70s		
Pennant	(E)	1	33.66s		
MiniGMG	(F)	1	6.10s		

Evaluation — Benchmark Description

Benchmark	ID	# Threads	Base Time	# Successful Compilations	# New Versions
RSBench	(A)	72	8.56s	32	
XSBench	(B)	1	75.13s	47	
PathFinder	(C)	1	363.50s	62	
CoMD	(D)	72	44.70s	49	
Pennant	(E)	1	33.66s	69	
MiniGMG	(F)	1	6.10s	16	

Evaluation — Benchmark Description

Benchmark	ID	# Threads	Base Time	# Successful Compilations	# New Versions
RSBench	(A)	72	8.56s	32	**9 (28.1%)**
XSBench	(B)	1	75.13s	47	**5 (10.6%)**
PathFinder	(C)	1	363.50s	62	**22 (35.5%)**
CoMD	(D)	72	44.70s	49	**13 (26.5%)**
Pennant	(E)	1	33.66s	69	**12 (17.4%)**
MiniGMG	(F)	1	6.10s	16	**4 (25.0%)**

Evaluation — Taken Opportunities

	Function Behavior			
	memory	exceptions	return value	linkage
(A)	5/7	2	0	4
(B)	4	0	0	1/3
(C)	15/23	14	0	2
(D)	3/4	1	0	0
(E)	18/37	9/14	8	33/37
(F)	3	1	0	2

Evaluation — Taken Opportunities

	Pointer Arguments				
	Alias	Escape	Usage	Alignment	Dereferenceable
(A)	19	0	11	11	19
(B)	1/2	0	1/2	11	11
(C)	16	15	12	16	16
(D)	2	0	2	1/2	2
(E)	66/78	71	77/79	53/85	46
(F)	5	0	4	5	5

Evaluation — Taken Opportunities

	Alignment		Deref.	Inv.	Taken vs. Total Optimistic
	Loaded Ptr.	Acc. Ptr.	Loaded Ptr.	Load	Opportunities
(A)	63/64	20/21	21	34/45	**225/240 (93.8%)**
(B)	33/34	9/10	10	28	**129/141 (91.5%)**
(C)	29/30	37/41	38	34/37	**264/299 (88.3%)**
(D)	61/71	25/26	26	32	**179/194 (92.3%)**
(E)	91/92	37	36/37	35/37	**610/689 (88.5%)**
(F)	132	44	44	25	**479 (100%)**

Evaluation — RSBench

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

Evaluation — XSBench

speedup · new "correct" version · existing "correct" version · tuning time

Speedup: 1, 1.05, 1.1, 1.15, 1.2

Tuning Time (s): 0, 100, 200, 300, 400

Compilations: 0, 5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60, 65, 70, 75, 80, 85, 90, 95

XSBench

Evaluation — PathFinder

speedup · new “correct” version · existing “correct” version · tuning time

Speedup: 0.95, 1, 1.05, 1.1, 1.15, 1.2

Tuning Time (s): 0, 1,000, 2,000, 3,000, 4,000

Compilations: 0, 20, 40, 60, 80, 100, 120, 140, 160, 180, 200, 220, 240

PathFinder

Evaluation — CoMD

speedup · new “correct” version · existing “correct” version · tuning time

Speedup: 0.94, 0.96, 0.98, 1, 1.02, 1.04, 1.06

Tuning Time (s): 0, 500, 1,000, 1,500, 2,000, 2,500

Compilations: 0, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 110, 120

CoMD

Evaluation — Pennant

speedup · new “correct” version · existing “correct” version · tuning time

Speedup: 0.2, 0.4, 0.6, 0.8, 1

Tuning Time (s): 0, 10,000, 20,000, 30,000, 40,000

10^0, 10^1, 10^2, 500

Compilations (log scale)

Pennant

Evaluation — MiniGMG

speedup · new "correct" version · existing "correct" version · tuning time

MiniGMG

Speedup: 1.1, 1.05, 1, 0.95

Tuning Time (s): 600, 400, 200, 0

Compilations: 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

Final Remarks & Conclusion

Interested? Try It Out!

Interested? Try It Out!

1) Get the code: `https://github.com/jdoerfert/PETOSPA`

Interested? Try It Out!

1) Get the code: `https://github.com/jdoerfert/PETOSPA`

2) Run the tuner: `./optimistic_tuner XSbench.json`

Interested? Try It Out!

1) Get the code: `https://github.com/jdoerfert/PETOSPA`

2) Run the tuner: `./optimistic_tuner XSbench.json`

3) Review suggestions:

```
clang -O3 --Rpass=optimistic-annotator xs_kernel.c
```

Interested? Try It Out!

1) Get the code: `https://github.com/jdoerfert/PETOSPA`

2) Run the tuner: `./optimistic_tuner XSbench.json`

3) Review suggestions:

```
clang -O3 --Rpass=optimistic-annotator xs_kernel.c
```

```
 In file included from xs_kernel.c:1: ./rsbench.h:94:16: remark: provide
better information on the memory effects of function 'fast_cexp', e.g.,
through '__attribute__((pure))' or '__attribute__((const))'
(assumed function speculative read-none).
complex double fast_cexp( double complex z );
```

Interested? Try It Out!

1) Get the code: `https://github.com/jdoerfert/PETOSPA`

2) Run the tuner: `./optimistic_tuner XSbench.json`

3) Review suggestions:

```
clang -O3 --Rpass=optimistic-annotator xs_kernel.c

 In file included from xs_kernel.c:1: ./rsbench.h:94:16: remark: provide
better information on the memory effects of function 'fast_cexp', e.g.,
through '__attribute__((pure))' or '__attribute__((const))'
(assumed function speculative read-none).
complex double fast_cexp( double complex z );
```

4) Don't forget to email me your findings ;)

Conclusion

The Compiler Black Box — Enabling Optimizations

```
>> 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;
   }
```

Conclusion

The Compiler Black Box — Enabling Optimizations

```
>> 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;
}
```

Annotation Opportunities — We utilize in LLVM

- 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

Conclusion

The Compiler Black Box — Enabling Optimizations

```
>> 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;
}
```

Annotation Opportunities — We utilize in LLVM

- 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

Evaluation — Benchmark Description

Benchmark	ID	# Threads	Base Time	# Successful Compilations	# New Versions
RSBench	(A)	72	8.56s	32	9 (28.1%)
XSBench	(B)	1	75.13s	47	5 (10.6%)
PathFinder	(C)	1	363.50s	62	22 (35.5%)
CoMD	(D)	72	44.70s	49	13 (26.5%)
Pennant	(E)	1	33.66s	69	12 (17.4%)
MiniGMG	(F)	1	6.10s	16	4 (25.0%)

Conclusion

The Compiler Black Box — Enabling Optimizations

```
>> 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;
  }
```

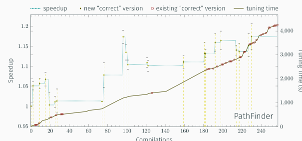
Annotation Opportunities — We utilize in LLVM

- 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

Evaluation — Benchmark Description

Benchmark	ID	# Threads	Base Time	# Successful Compilations	# New Versions
RSBench	(A)	72	8.56s	32	9 (28.1%)
XSBench	(B)	1	75.13s	47	5 (10.6%)
PathFinder	(C)	1	363.50s	62	22 (35.5%)
CoMD	(D)	72	44.70s	49	13 (26.5%)
Pennant	(E)	1	33.66s	69	12 (17.4%)
MiniGMG	(F)	1	6.10s	16	4 (25.0%)

Evaluation — PathFinder



Current & Future Work

Current & Future Work

- Allow encoding of all opportunities in the source.

Current & Future Work

- Allow encoding of all opportunities in the source.
- Rewrite source code automatically for sound choices.

Current & Future Work

- Allow encoding of all opportunities in the source.
- Rewrite source code automatically for sound choices.
- Work with multiple compilers, e.g., for validation and portability.

Potentially Overflowing Computations

```
if (u <ᵤ u -   1)
  foo();
if (i <ₛ i +   1)
  bar();
```

Potentially Overflowing Computations

```
if (u <_u u -   1)
  foo();
if (i <_s i +   1)
  bar();
```

We want this concise code:

```
bar();
```

Potentially Overflowing Computations

```
if (u <u u -   1)
  foo();
if (i <s i +   1)
  bar();
```

(In the code as printed, the first `<` has a subscript u and the second `<` has a subscript s: $<_u$ and $<_s$.)

Is it optimized?

We want this concise code:

```
bar();
```

Potentially Overflowing Computations

```
if (u <_u u -_nuw 1)
  foo();
if (i <_s i +_nsw 1)
  bar();
```

Is it optimized?

Yes, with `nsw/nuw`!

We want this concise code:

```
bar();
```

```
unsigned Mode;

void baz(int *A);

void foo(int *A) {

  if (Mode == MODE_A)
    ...
}

void bar(int *B) {

  if (Mode == MODE_B)
    ...
}
```

```
unsigned Mode;

void baz(int *A);

void foo(int *A) {
  __builtin_assume(Mode == MODE_A);
  if (Mode == MODE_A)
    ...
}

void bar(int *B) {

  if (Mode == MODE_B)
    ...
}
```

Compile Time Improvements — Caching by Kind & Name

```
unsigned Mode;

void baz(int *A);

void foo(int *A) {
  __builtin_assume(Mode == MODE_A);
  if (Mode == MODE_A)
    ...
}

void bar(int *B) {
  __builtin_assume(Mode == MODE_A);
  if (Mode == MODE_B)
    ...
}
```

Compile Time Improvements — Caching by Kind & Name

```
unsigned Mode;

void baz(int *A);

void foo(int *A) {
  __builtin_assume(Mode == MODE_A);
  if (Mode == MODE_A)
    ...
}

void bar(int *B) {
  __builtin_assume(Mode == MODE_A);
  if (Mode == MODE_B)
    ...
}
```

Compile Time Improvements — Caching by Kind & Name

```
unsigned Mode;

void baz(__attribute__((noescape)) int *A);

void foo(__attribute__((noescape)) int *A) {
  __builtin_assume(Mode == MODE_A);
  if (Mode == MODE_A)
    ...
}

void bar(int *B) {
  __builtin_assume(Mode == MODE_A);
  if (Mode == MODE_B)
    ...
}
```

Configuration File — The User Provided Input

```json
{
    "name": "test",
    "source_files": [
        {
            "path": "main.c",
            "options": ["-ffast-math", "-O3"],
            "only_functions": ["foo", "bar"]
        }
    ],
    "executable": "./a.out",
    "make_cmd": "make",
    "verify_cmd": "verify.sh"
}
```

Annotation Opportunities per Insertion Point

(F)
(E)
(D)
(C)
(B)
(A)

0 10 20 30 40 50 60 70 80 90 100

1 2 3 4 5 6 7 8 9 10 11 12 13 14

Additional Features

Additional Features

- Optimization choices can be restricted to kinds, e.g., test no-alias and no-escape annotations only.

Additional Features

- Optimization choices can be restricted to kinds, e.g., test no-alias and no-escape annotations only.
- We use multiple (14) insertion points for optimistic annotations throughout the optimization pipeline.

Additional Features

- Optimization choices can be restricted to kinds, e.g., test no-alias and no-escape annotations only.
- We use multiple (14) insertion points for optimistic annotations throughout the optimization pipeline.
- Optimization choices can be encoded in the program for search resumeability, annotation of non-source properties, and as alternative to command line encoding.