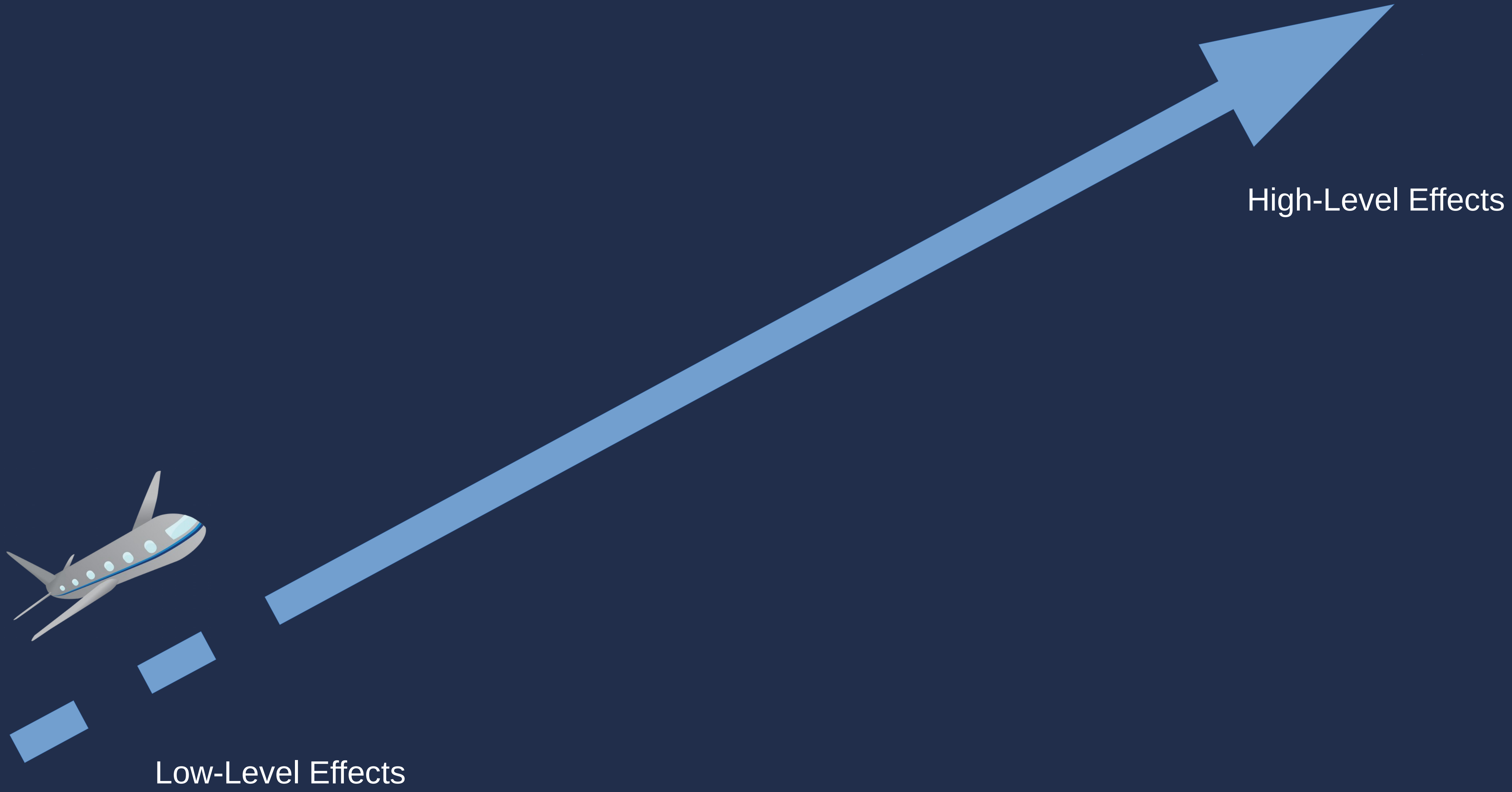
DOE Proxy Apps – Clang/LLVM vs. the World!

Hal Finkel, Brian Homerding, Michael Kruse
EuroLLVM 2018


High-Level Effects
Low-Level Effects

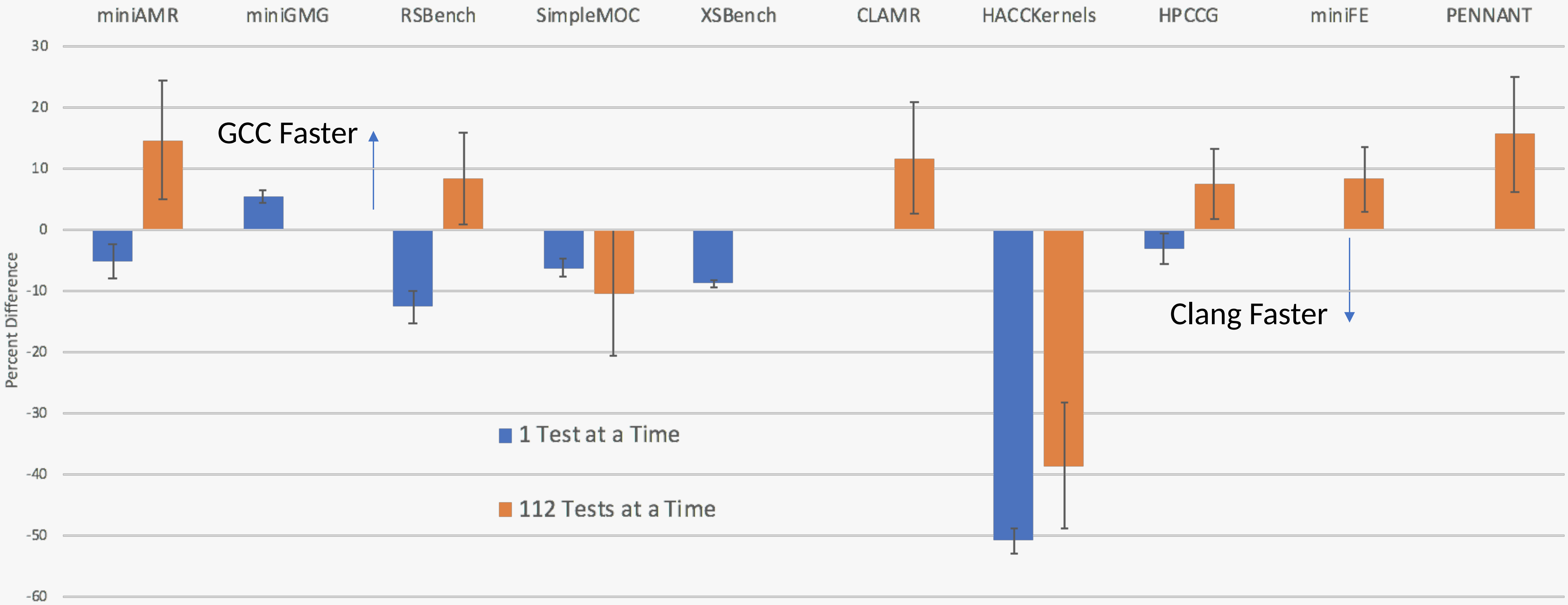
Many Good Stories Start with Some Source of Confusion...

Why do *you* think Clang/LLVM is doing better than *I* do?

Test Suite Analysis Methodology

- Collect 30 samples of execution time of test suite using lnt with both clang 7 and GCC 7.3 using all threads including hyper-threading (112 on Skylake run and 88 on Broadwell run) (Noisy System)
- Compare with 99.5% confidence level using ministat
- Collect 30 additional samples for each compiler with only a single thread being used (Quiet System)
- Compare with 99.5% confidence level using ministat
- Look at the difference between compiler performance with different amounts of noise on the system
- Removed Some Outliers (Clang 20,000% faster on Shootout-C++-nestedloop)

Subset of DOE Proxies

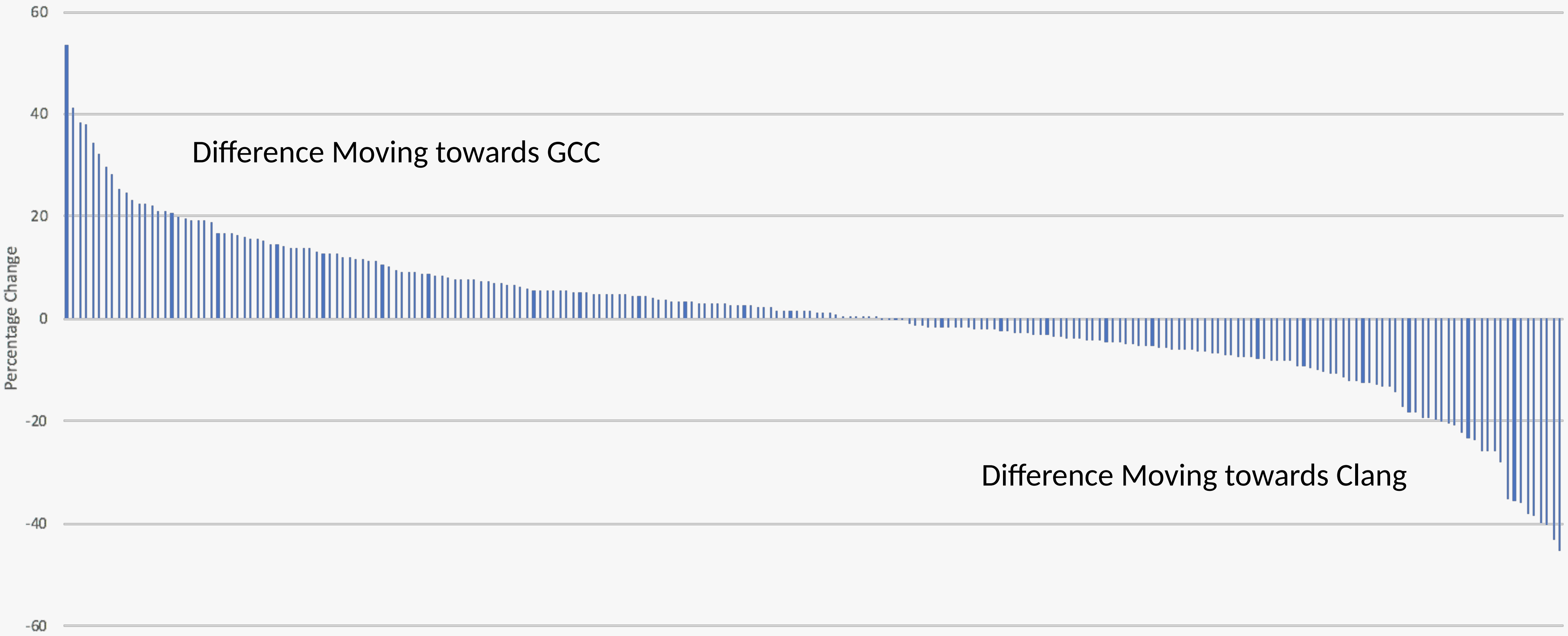


Argonne NATIONAL LABORATORY

Several of the DOE Proxy Apps are Interesting

- MiniAMR, RSBench and HPCCG jump the line and GCC begins to outperform
- PENNANT, MiniFE and CLAMR show GCC outperforming when there was no difference on a quite system
- XSBench shows Clang outperforming on a quiet system and no difference on a noisy system (memory latency sensitive)

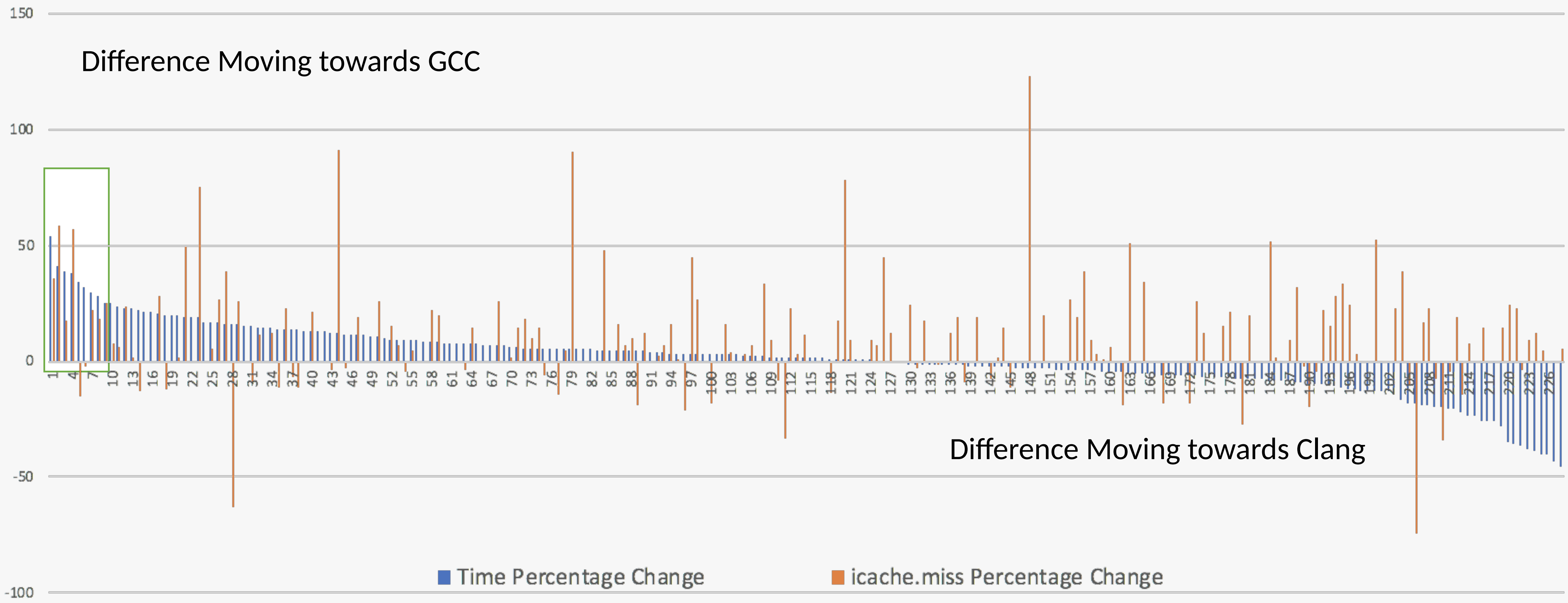
= Statistical Difference on 112 Threads – Statistical Difference on 1 Thread



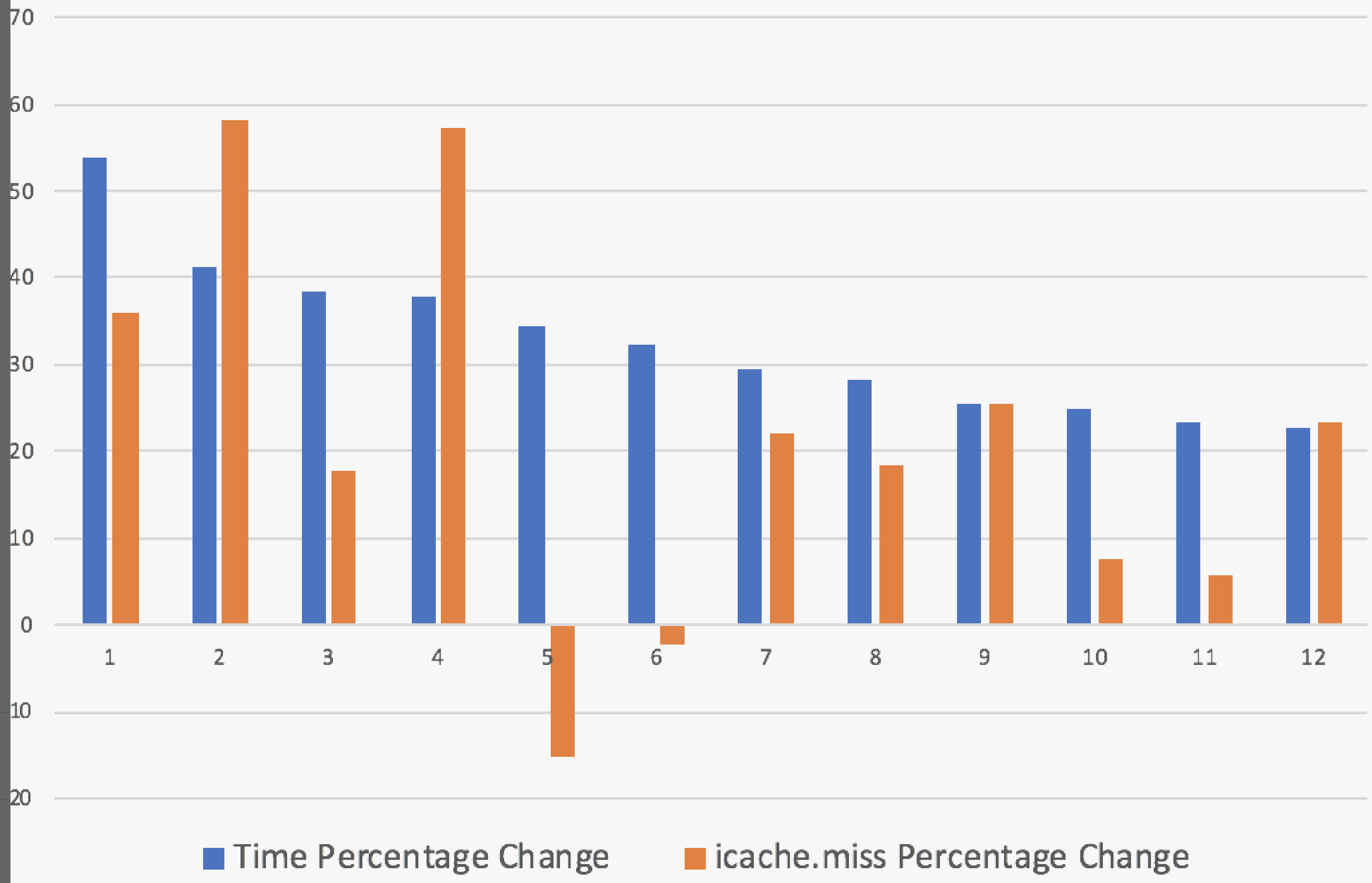
What is causing the statistical difference?

- Instruction Cache Misses?
- Rerun methodology collecting performance counters 30 Samples each compiler for both quiet and noisy system

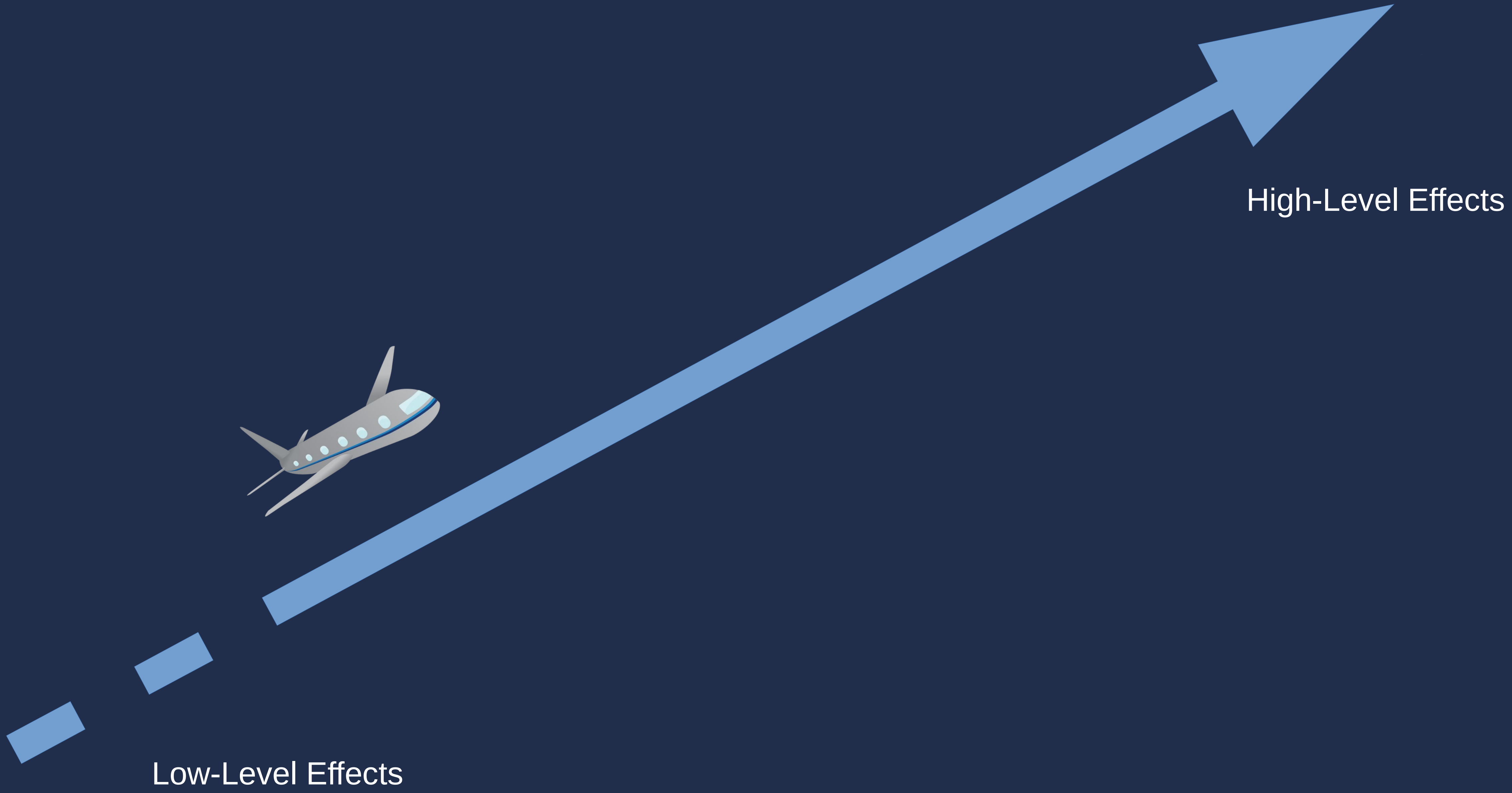
Instruction Cache Miss Data Added



Top 12 tests where performance trends towards GCC on noisy system



- Instruction cache misses do appear to explain some of the cases but is not the only relevant factor.


High-Level Effects
Low-Level Effects

RSBench Proxy Application
Significant amount of work in math library

```
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 7

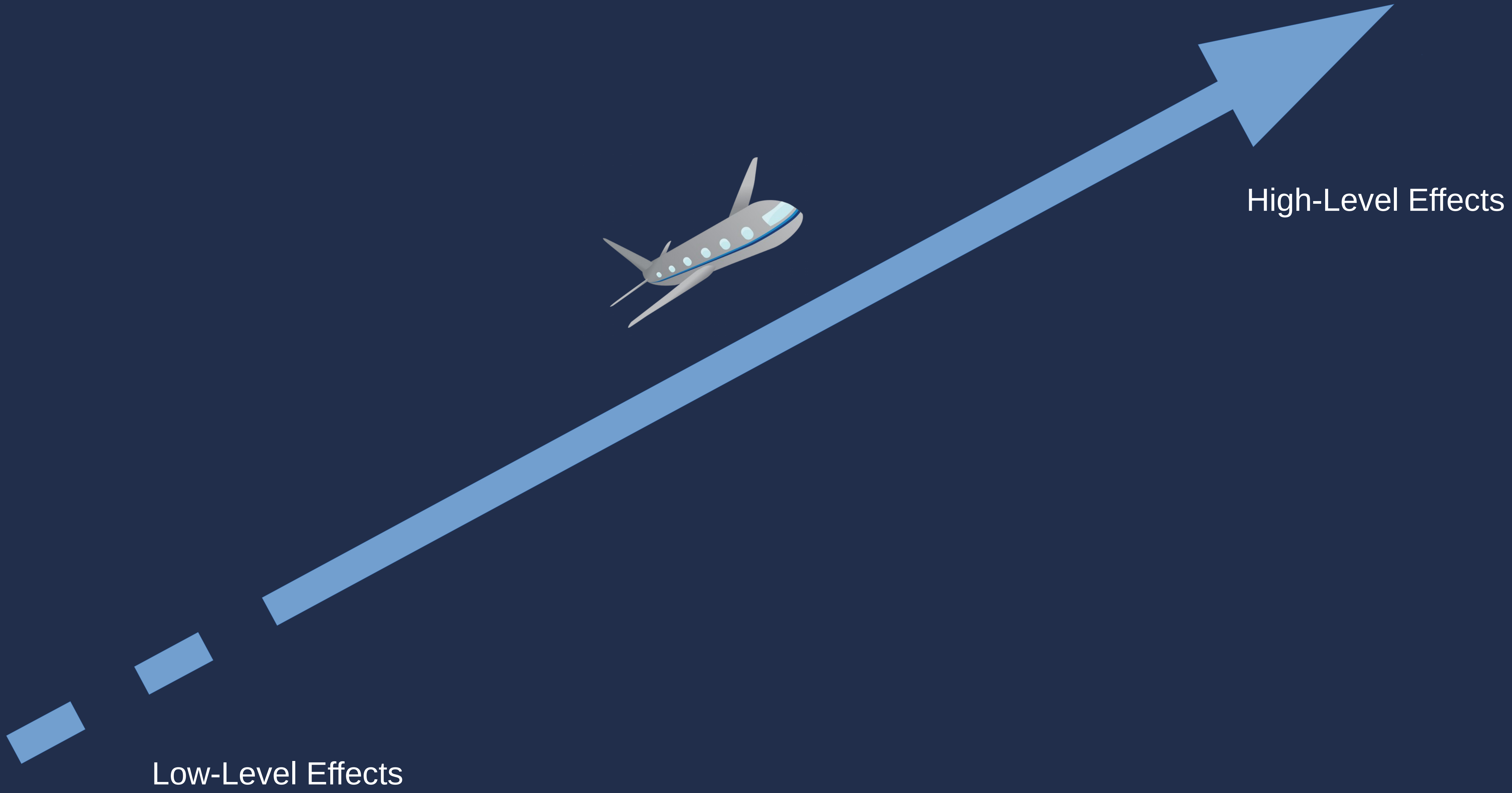
```
callq   cos
vmovsd  %xmm0, 8(%rsp)              # 8-byte Spill
vmovsd  56(%rsp), %xmm0             # 8-byte Reload
                                    # xmm0 = mem[0],zero
callq   sin
vmovsd  .LCPI2_4(%rip), %xmm1       # xmm1 = mem[0],zero
vmovapd %xmm1, %xmm2
```

GCC 7.3

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

For This, We Have A Plan: Modelling write-only errno

- Missed SimplifyLibCall
- Current limitations with representing write-only functions
- Write only attribute in clang
- Marking math functions as write only
- Special case that sin and cos affect memory in the same way


High-Level Effects
Low-Level Effects

Compiler Specific Pragmas

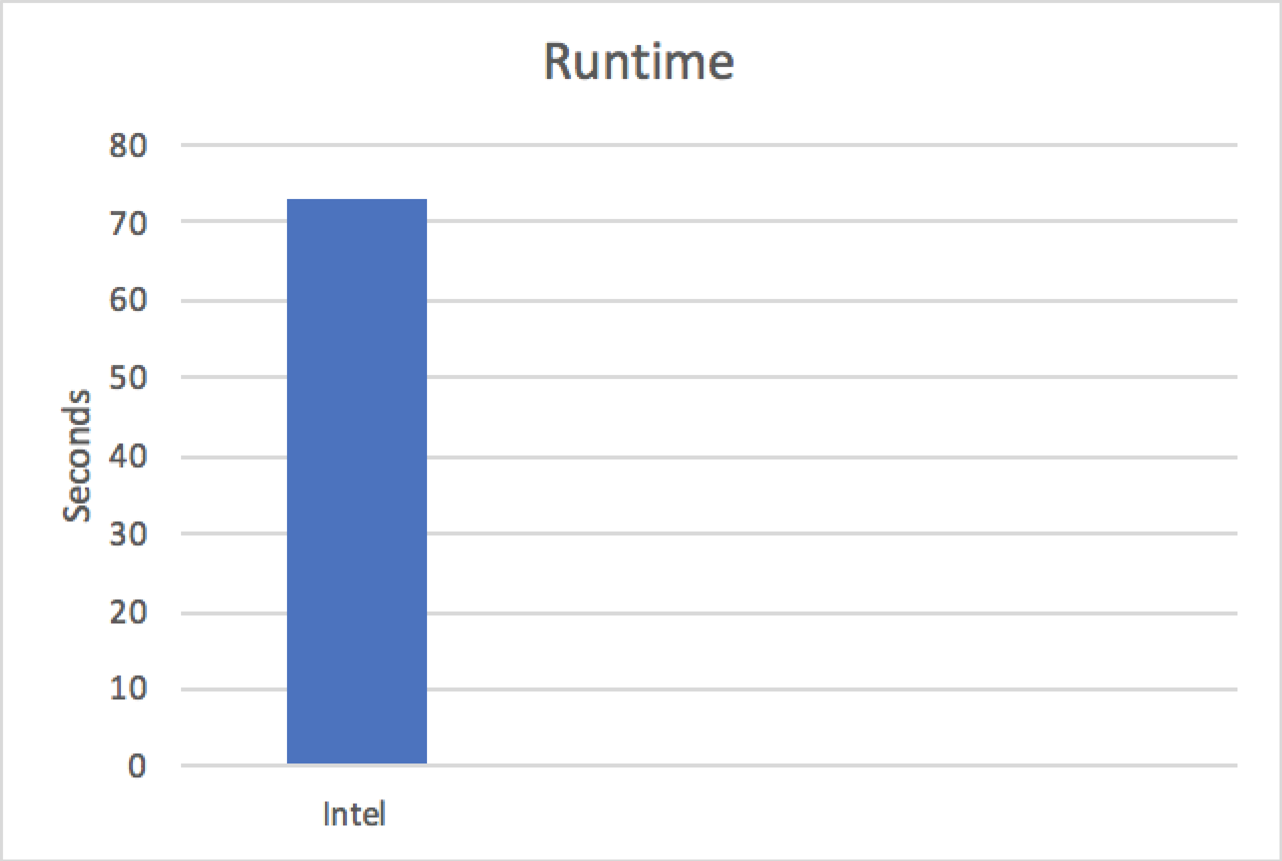
- #pragma ivdep
- #pragma loop_count(15)
- #pragma vector nontemporal

- Clear mapping of to Clang pragmas?

- Not always just specific pragmas

```
#ifdef INTEL
#pragma simd
#elif defined IBM
#pragma simd_level(10)
#endif
```

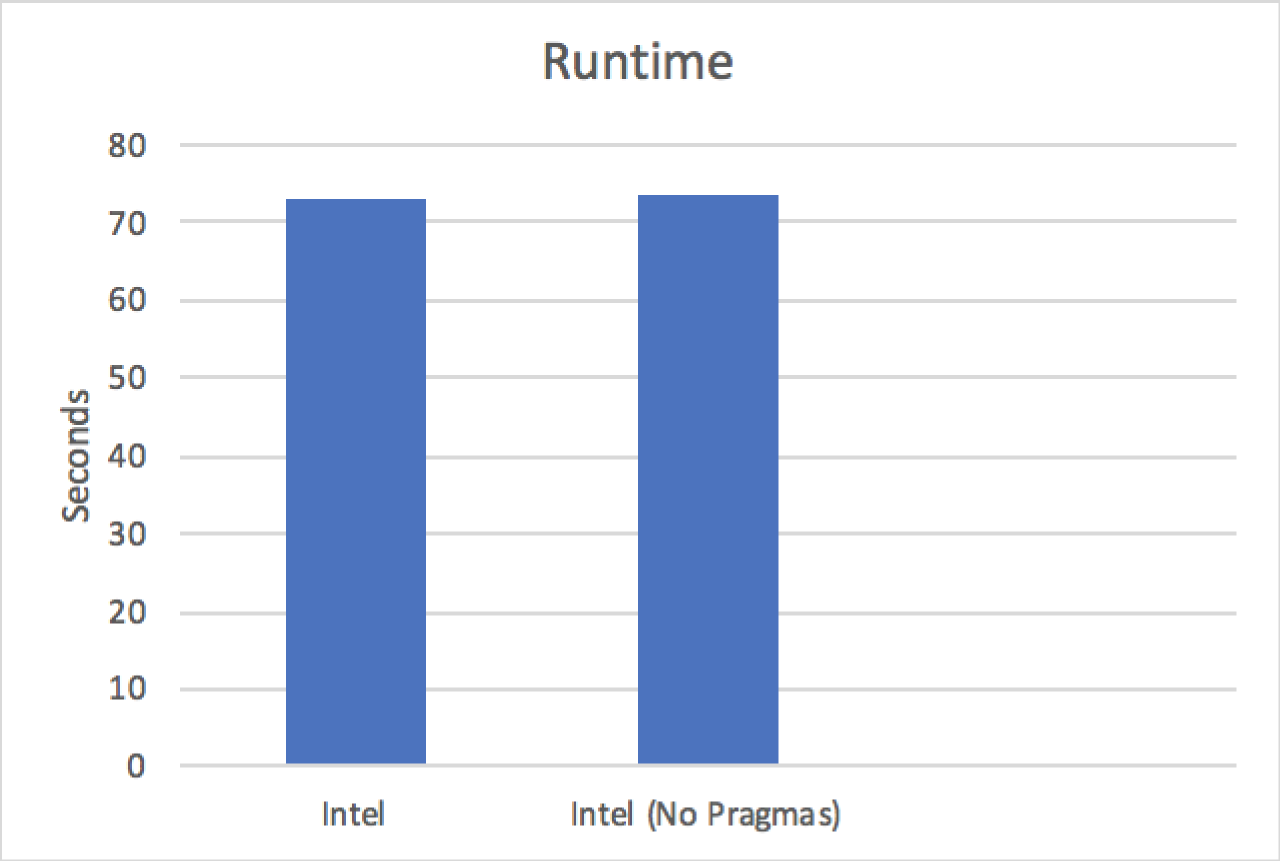
MiniFE Proxy Application / openmp-opt
./miniFE.x –nx 420 –ny 420 –nz 420



- Compiler Specific Pragmas

```
#pragma loop_count(15)
#pragma vector nontemporal
```

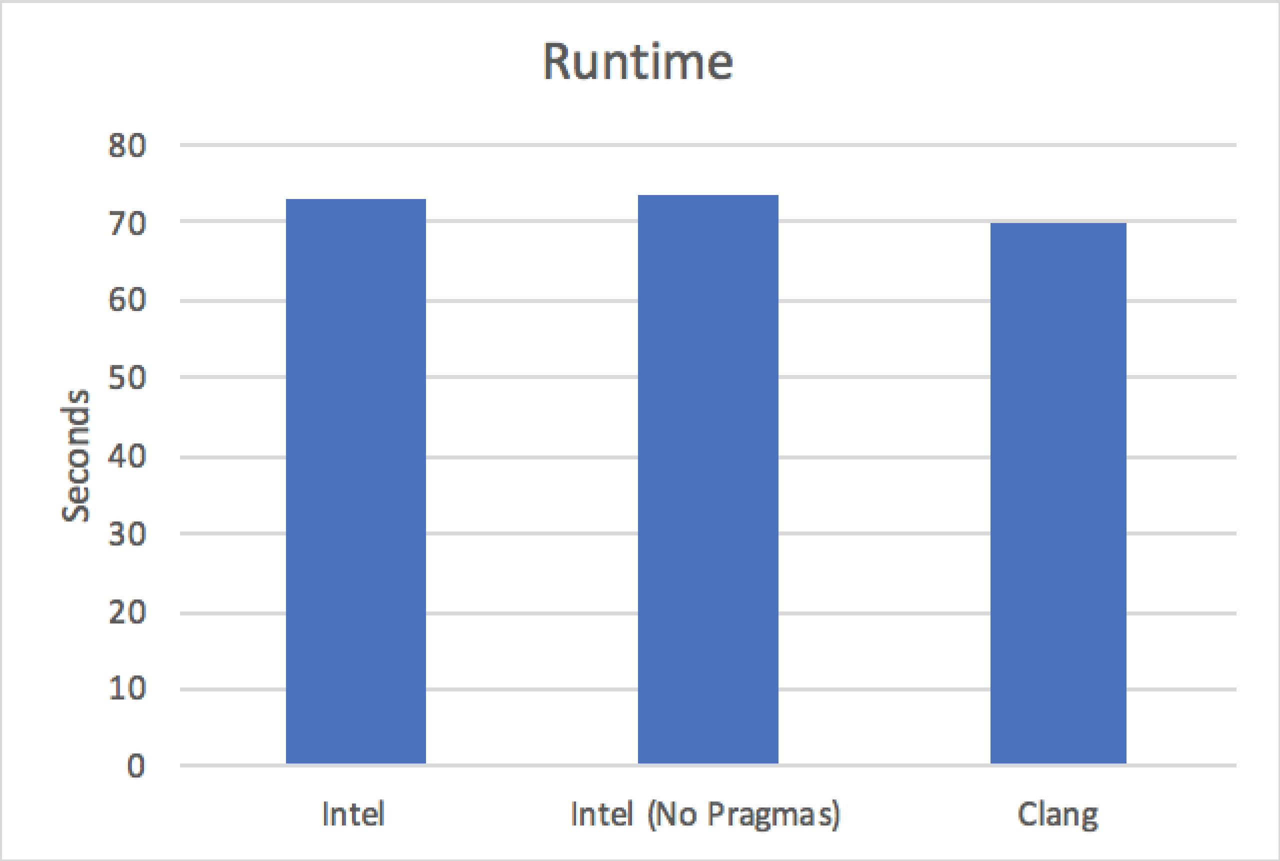
MiniFE Proxy Application / openmp-opt
./miniFE.x –nx 420 –ny 420 –nz 420



- Compiler Specific Pragmas

```
#pragma loop_count(15)
#pragma vector nontemporal
```

MiniFE Proxy Application / openmp-opt ./miniFE.x –nx 420 –ny 420 –nz 420



- Compiler Specific Pragmas

```
#pragma loop_count(15)
#pragma vector nontemporal
```

Compiler Specific Pragmas

- Intel Compiler shows little to no performance gain from #pragmas for fully optimized applications investigated thus far

- #pragma loop_count(15)
- #pragma ivdep
- #pragma vector nontemporal

- Is there a potential benefit from this additional information that is not yet realized? Were the pragmas needed in a previous version and not now? Were they needed in the full application but not in the proxy?

LCALS “Livermore Compiler Analysis Loop Suite”

Subset A:

- Loops representative of those found in application codes

Subset B:

- Basic loops that help to illustrate compiler optimization issues

Subset C:

- Loops extracted from “Livermore Loops coded in C” developed by Steve Langer, which were derived from the Fortran version by Frank McMahon

Google Benchmark Library

- Runs each micro-benchmark a variable amount of times and reports the mean. The library controls the number of iterations.

- Provides additional support for specifying different inputs, controlling measurement units, minimum kernel runtime, etc…

- Did not match lit's one test to one result reporting

Expanding lit

- Expand the lit Result object to allow for a one test to many result model

```
def addMicroResult(self, name, microResult):
    """
    addMicroResult(microResult)

    Attach a micro-test result to the test result, with the given name and
    result.  It is an error to attempt to attach a micro-test with the
    same name multiple times.

    Each micro-test result must be an instance of the Result class.
    """
    if name in self.microResults:
        raise ValueError("Result already includes microResult for %r" % (
               name,))
    if not isinstance(microResult, Result):
        raise TypeError("unexpected MicroResult value %r" % (microResult,))
    self.microResults[name] = microResult
```

Expanding lit

- The test suite can now use lit report individual kernel timings based on the mean of many iterations of the kernel

test-suite/MicroBenchmarks

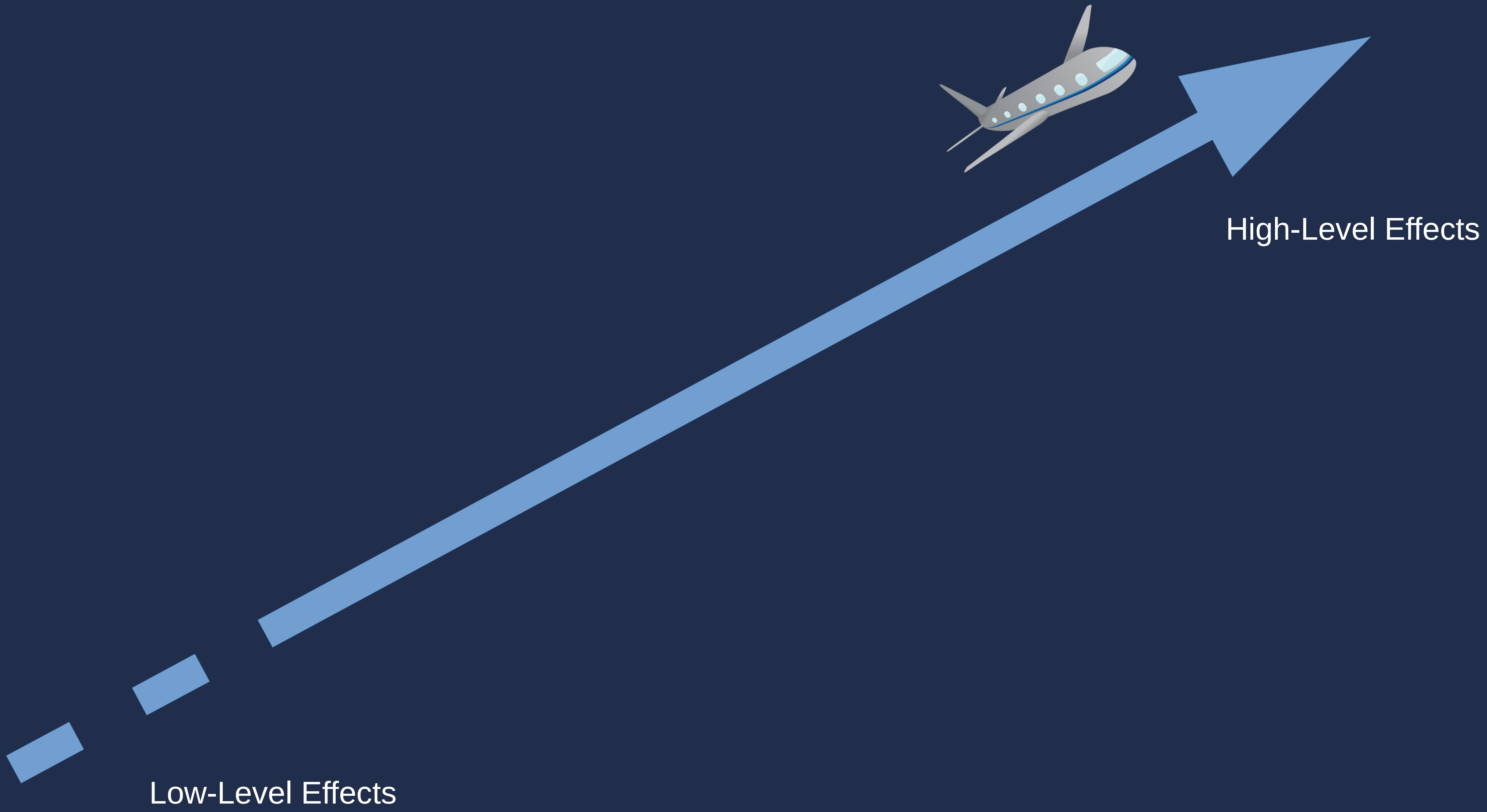
```
PASS: test-suite :: MicroBenchmarks/LCALS/SubsetBRawLoops/lcalsBRaw.test
(1 of 1)
********** TEST 'test-suite ::
MicroBenchmarks/LCALS/SubsetBRawLoops/lcalsBRaw.test' RESULTS **********
MicroBenchmarks: 12
compile_time: 6.9610
hash: "5075a3ae907cf9631cdc4cf8401cbfb3"
link_time: 0.0426
**********
*** MICRO-TEST: BM_IF_QUAD_RAW/171
    exec_time:  2.6995
*** MICRO-TEST: BM_IF_QUAD_RAW/44217
    exec_time:  698.8880
*** MICRO-TEST: BM_IF_QUAD_RAW/5001
    exec_time:  78.9838
*** MICRO-TEST: BM_INIT3_RAW/171
    exec_time:  0.2248
*** MICRO-TEST: BM_INIT3_RAW/44217
    exec_time:  168.0970
*** MICRO-TEST: BM_INIT3_RAW/5001
    exec_time:  15.1119
*** MICRO-TEST: BM_MULADDSUB_RAW/171
    exec_time:  0.4491
*** MICRO-TEST: BM_MULADDSUB_RAW/44217
    exec_time:  169.6760
*** MICRO-TEST: BM_MULADDSUB_RAW/5001
    exec_time:  16.1443
*** MICRO-TEST: BM_TRAP_INT_RAW/171
    exec_time:  2.0922
*** MICRO-TEST: BM_TRAP_INT_RAW/44217
    exec_time:  540.9620
*** MICRO-TEST: BM_TRAP_INT_RAW/5001
    exec_time:  61.1846
```

LLVM Test Suite MicroBenchmarks

- Write benchmark code using the Google Benchmark Library
  https://github.com/google/benchmark
- Add test code into test-suite/MicroBenchmarks
- Link executable in test's CMakeLists to benchmark library

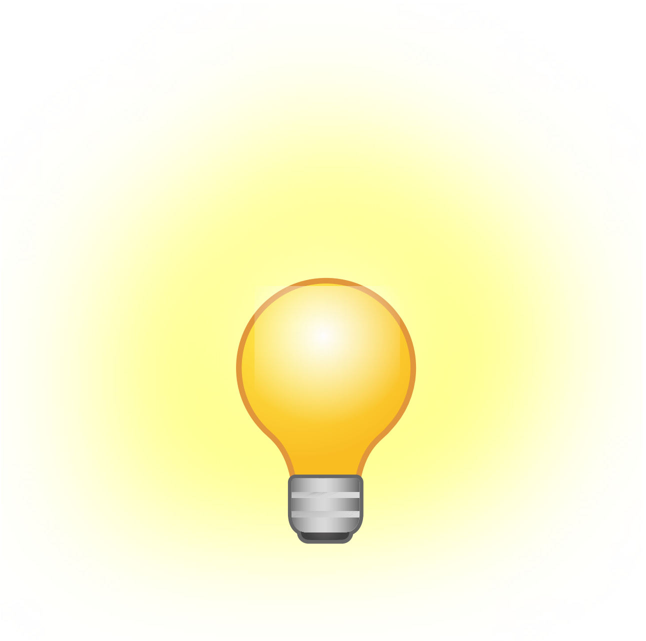
```
target_link_libraries(lcalsALambda benchmark)
```

- lit.local.cfg in test-suite/MicroBenchmarks will include the microBenchmark module from test-suite/litsupport


High-Level Effects
Low-Level Effects

And Now To Talk About Loops and Directives...

Some plans for a new loop-transformation framework in LLVM...



EXISTING LOOP TRANSFORMATIONS

Loop Transformation #pragmas are Already All Around

gcc

`#pragma unroll 4` [also supported by clang, icc, xlc]

clang

```
#pragma clang loop distribute(enable)
#pragma clang loop vectorize_width(4)
#pragma clang loop interleave(enable)
```
`#pragma clang loop vectorize(assume_safety)` [undocumented]

icc

```
#pragma ivdep
#pragma distribute_point
```

msvc

```
#pragma loop(hint_parallel(0))
```

xlc

```
#pragma unrollandfuse
#pragma loopid(myloopname)
#pragma block_loop(50, myloopname)
```

OpenMP/OpenACC

```
#pragma omp parallel for
```

SYNTAX

Current syntax:

– `#pragma clang loop transformation(option) transformation(option) ...`

– Transformation order determined by pass manager
– Each transformation may appear at most once
– LoopDistribution results in multiple loops, to which one apply follow-ups?

Proposed syntax:

– `#pragma clang loop transformation option option(arg) ...`

– One `#pragma` per transformation
– Transformations stack up
– Can apply same transformation multiple times
– Resembles OpenMP syntax

AVAILABLE TRANSFORMATIONS

Ideas, to be Implemented Incrementally

```
#pragma clang loop stripmine/tile/block
#pragma clang loop split/peel/concatenate   [index domain]
#pragma clang loop specialize                   [loop versioning]
#pragma clang loop unswitch
#pragma clang loop shift/scale/skew         [induction variable]
#pragma clang loop coalesce
#pragma clang loop distribute/fuse
#pragma clang loop reverse
#pragma clang loop move
#pragma clang loop interchange
#pragma clang loop parallelize_threads/parallelize_accelarator
#pragma clang loop ifconvert
#pragma clang loop zcurve
#pragma clang loop reschedule algorithm(pluto)
#pragma clang loop assume_parallel/assume_coincident/assume_min_depdist
#pragma clang loop assume_permutable
#pragma clang data localize                 [copy working set used in loop body]
...
```

LOOP NAMING

Ambiguity when Transformations Result in Multiple Loops

```
#pragma clang loop vectorize width(8)
#pragma clang loop distribute
for (int i = 1; i < n; i+=1) {
  A[i] = A[i-1] + A[i];
  B[i] = B[i] + 1;
}
```

```
#pragma clang loop vectorize width(8)
for (int i = 1; i < n; i+=1)
  A[i] = A[i-1] + A[i];
#pragma clang loop vectorize width(8)
for (int i = 1; i < n; i+=1)
  B[i] = B[i] + 1;
```

[<= not vectorizable]

LOOP NAMING

Solution: Loop Names

```
#pragma clang loop(B) vectorize width(8)
#pragma clang loop distribute              [← applies implicitly on next loop]
for (int i = 1; i < n; i+=1) {
  #pragma clang block id(A)
  { A[i] = A[i-1] + A[i]; }
  #pragma clang block id(B)
  { B[i] = B[i] + 1; }
}

#pragma clang loop id(A)                   [← implicit name from loop distribution]
for (int i = 1; i < n; i+=1)
  A[i] = A[i-1] + A[i];
#pragma clang loop vectorize width(8)
#pragma clang loop id(B)                   [← implicit name from loop distribution]
for (int i = 1; i < n; i+=1)
  B[i] = B[i] + 1;
```

OPEN QUESTIONS

Is
`#pragma clang loop parallelize_threads`
different enough from
`#pragma omp parallel for`
to justify its addition?

How to encode different parameters for different platforms?

Is it possible to use such #pragmas outside of the function the loop is in?

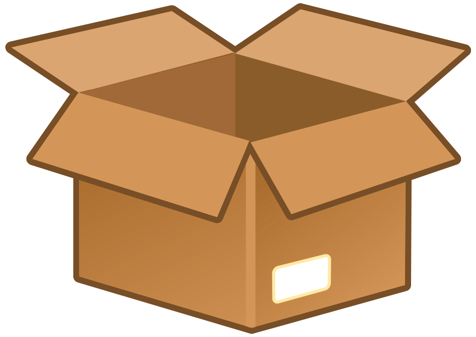
- Would like to put the source into a different file, which is then #included

Does the location of a #pragma with a loop name have a meaning?

Implementing These Using Polly...

As you might imagine, Polly's infrastructure can make this relatively easy in many cases…

But there are challenges!



BASED ON SCOP-REGIONS

Restrictions on SCoPs Apply

- Only Single-Entry Single-Exit (SESE) regions

```
#pragma clang loop transform
for (int i = 0; i < n; i+=1) {
  if (residual < 1e-8) break;
  ...
}
```

- Non-affine loop bounds

```
#pragma clang loop transform
for (int i = 0; i < rows; i+=1)
  for (int j = 0; j < row[i]->cols; j+=1)
    { ... }
```

- Non-affine control flow is atomic

```
#pragma clang loop distribute
for (int i = 0; i < n; i+=1)
  if (flag[i]) {
    A[i] = A[i] + 1;
    B[i] = B[i] * 2;
  }
```

- Statically infinite loops

```
#pragma clang loop transform
for (int i = 0; i < n; i+=1) {
  if (c) while (true) { ... }
  ...
}         [LoopInfo considers the while-loop outside the outer loop, but for RegionInfo it is inside]
```

BASED ON SCOP-REGIONS

Restrictions on SCoPs Apply (cont.)

- No exceptions (incl. mayThrow() or invoke)

```
#pragma clang loop transform
for (int i = 0; i < n; i+=1) {
  if (c < 0.0) throw std::invalid_argument("Must be non-negative");
  ...
}
```

- No VLAs inside loops

```
#pragma clang loop transform
for (int i = 0; i < n; i+=1) {
  double tmp[i];
  ...
}
```

- Complexity limits

```
#pragma clang loop transform
for (int i = 0; i < n; i+=1) {
  if (arg[0] != 5 && arg[1] != 6 && arg[2] != 7 && arg[3] != 8 && ...)
    {...}
}
```

- Checkable aliasing

```
double **A;
#pragma clang loop transform
for (int i = 0; i < n; i+=1)
  A[i][k] = ...;
```

- ....
- Even for always-safe transformations (e.g. unrolling), these SCoP-properties are required

BASED ON SCOP-REGIONS

More Restrictions, But al Least We Can Do Something About These

- Profitability heuristic still applies

```
#pragma clang loop unroll
for (int i = 1; i < n; i+=1)
  A[i] = A[i-1];
```

[Polly's profitability heuristics thinks there's nothing that can be done here]

- Always detect and codegen the max compatible region

```
for (int i = 0; i < rows; i+=1) {
  #pragma clang loop transform
  for (int j = 0; j < cols; j+=1)
    { ... }
}
```

[Even if only the inner loop is transformed, the outer loops is processed as well]

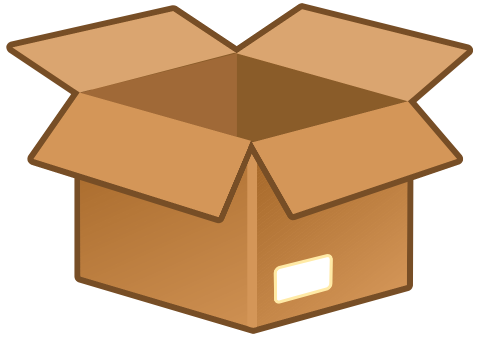
- Unpredictable loop bodies (e.g.: function calls that touch arbitrary memory)

```
#pragma clang loop transform
for (int i = 0; i < n; i+=1)
  printf("i = %d\n", i);
```

- Solution:
  Detect SCoPs differently in the presence of user-directed transformations

So What Do We Want To Do?

Create a modular infrastructure suitable for use by other transformations...



LOOP OPTIMIZATION FRAMEWORK

A Vision

- Do we want loop transformations in LLVM?
  - Question already answered: LoopInterchange, LoopDistribute, ...
  - If we have one, should be as good as possible
- Issues with the current pipeline:
  - Every transformation pass applies its own loop versioning
  - Each has its own dependency analysis

    [e.g. LoopInterchange's DependencyAnalysis only recently received some love]
  - Polly's aforementioned restrictions for SCoPs
  - Polly's pass model not supported by LLVM's pass manager

    [Polly has state that is not contained in the IR => lost at pass manager's will]
  - Polly is based on RegionInfo, other passes are LoopInfo-based
  - Polly assumes its loop versioning will be applied, therefore not directly usable as analysis by non-Polly passes

LOOP OPTIMIZATION FRAMEWORK

A Vision

- Source

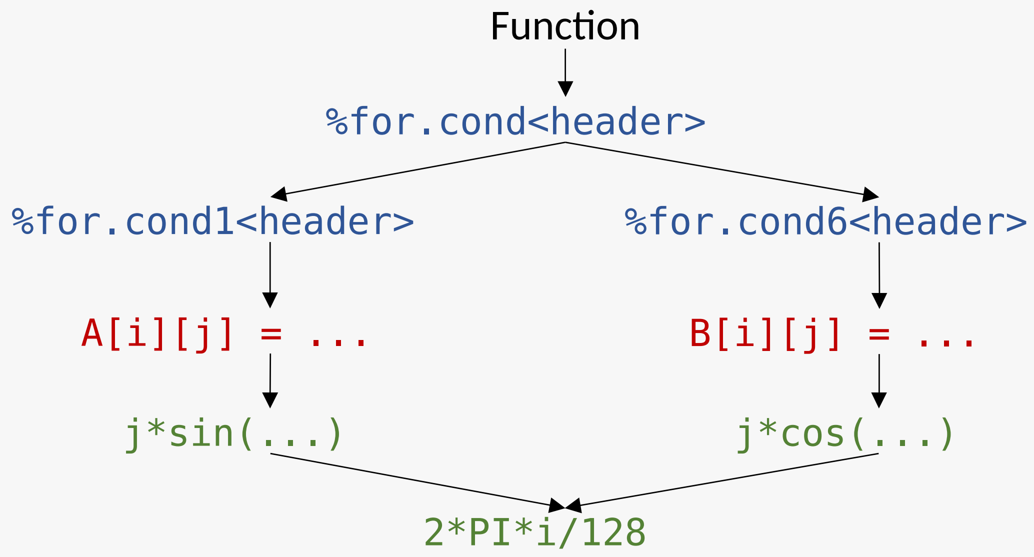
```
for (int i = 0; i < 128; i+=1) {
  for (int j = 0; j < 128; j+=1) A[i][j] = j*sin(2*PI*i/128);
  for (int j = 0; j < 128; j+=1) B[i][j] = j*cos(2*PI*i/128);
}
```

- ⇒ LoopInfo tree

```
Loop at depth 1 containing: %for.cond<header><exiting>,%for.body,%for.cond1,%for.cond.cleanup3,%for.end,%for.cond6,%for.cond.cleanup8,...
    Loop at depth 2 containing: %for.cond1<header><exiting>,%for.body4,%for.inc<latch>
    Loop at depth 2 containing: %for.cond6<header><exiting>,%for.body9,%for.inc10<latch>
```

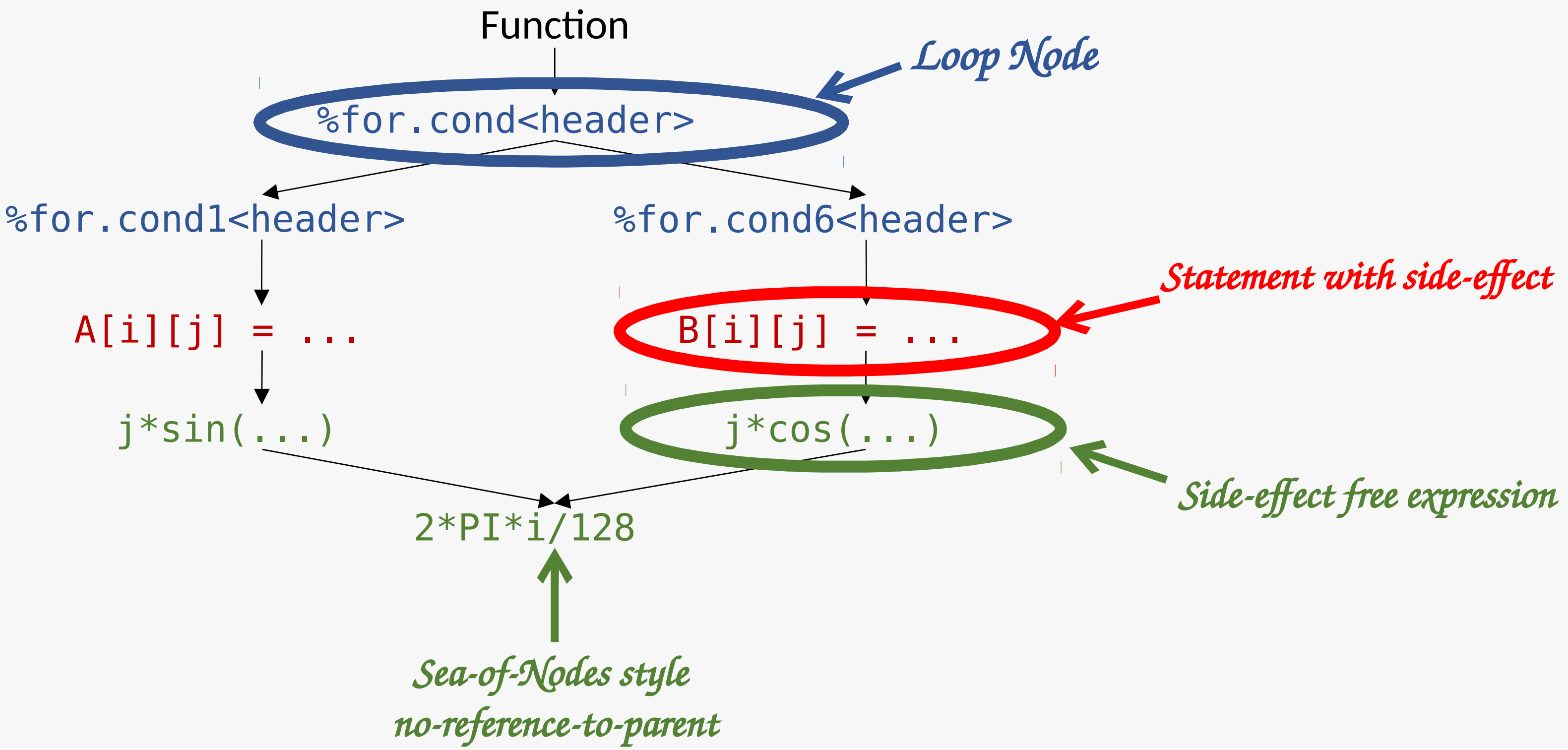
- ⇒ Loop AST (only precondition: no irreducible loops)

[irreducible loops can be transformed into reducible ones]



LOOP OPTIMIZATION FRAMEWORK

Loop Tree/DAG



[no-side-effect llvm::instructions best located where they are used;
(part of) Polly's DeLICM is about this. Sea-of-Nodes has it implicitly]

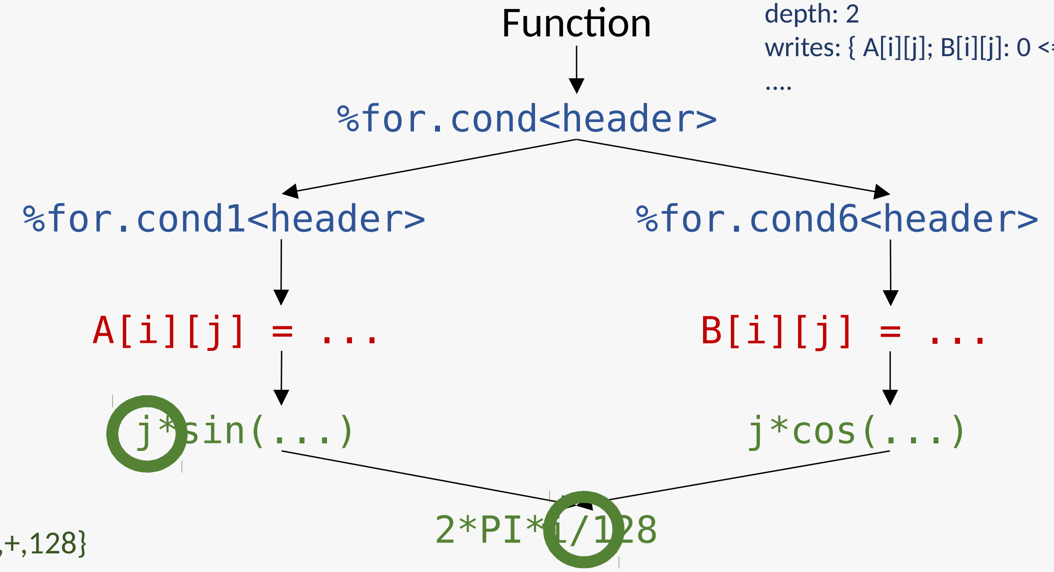
[consider "Equality Saturation: A New Approach to Optimization"]

LOOP OPTIMIZATION FRAMEWORK

Subtree analysis

isParallel: yes
idiom: ArrayInitialization
depth: 1
writes: [i] -> { A[i][j] : 0 <= i < 128 }
hasOtherAccesses: no
….

isParallel: yes
idiom: ArrayInitialization
depth: 2
writes: { A[i][j]; B[i][j]: 0 <= i,j < 128 }
….



isParallel: yes
idiom: ArrayInitialization
depth: 1
….

mayThrow: no
kills: [i,j] -> { A[i][j] }
writes: [i,j] -> { A[i][j] }
…

mayThrow: no
kills: [i,j] -> { B[i][j] }
defines: [i,j] -> { B[i][j] }
speculatable: no
idempotent: yes
…

- llvm::Value representation: %j
- SCEV representation: <%for.cond1>{0,+,128}
- Polyhedral Value Analysis: { [i,j] -> j }

- llvm::Value representation: %i
- SCEV representation: <%for.cond>{0,+,128}
- Polyhedral Value Analysis: { [i,j] -> i }

LOOP TRANSFORMATION FRAMEWORK

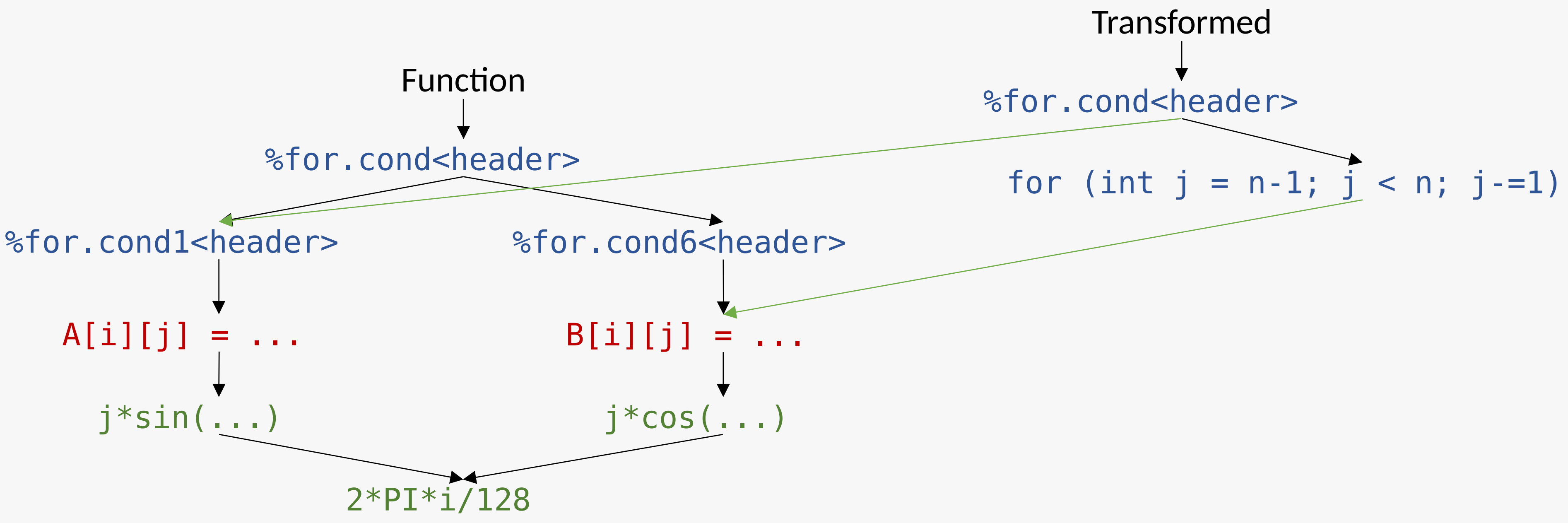
Loop Idioms

- Examples for loop idioms:
  - memcpy
  - memset
  - Array Initialization (only writes to array, no reads)
  - Pointwise (reads from A[i][j], writes to C[i][j])
  - Stencil (→ apply overlap/diamond/hybrids/... tiling)
  - Reduction
  - Convolution
  - Matrix-Multiplication (→ apply BLIS optimization/call BLAS library func)
    - Any code with similar structure
  - ...

LOOP OPTIMIZATION FRAMEWORK

Reusable Subtees

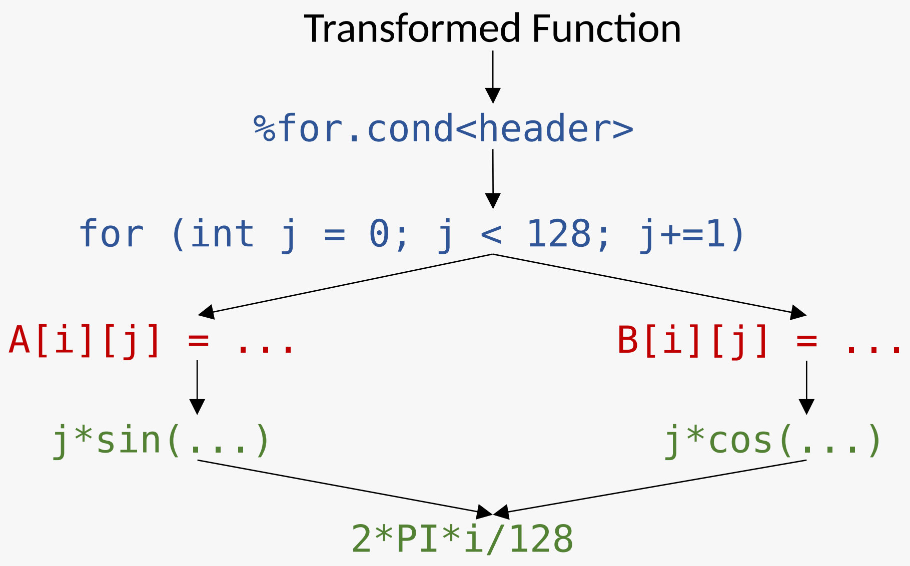
- Multiple code versions exist at the same time (like VPlan)
- Cheap copy using red/green-trees



LOOP OPTIMIZATION FRAMEWORK

Transformation: Loop Fusion

- Change tree, reuse subtrees
- Annotate nodes with assumptions under which its execution is correct



isParallel: yes
idiom: ArrayInitialization
assumptions:
- A[i][0..127] noalias B[i][0..127]

writes: [i] -> { A[i][j]; B[i][j] : 0 <= j < 128 }
...

LOOP OPTIMIZATION FRAMEWORK

More Transformations

- Apply transformation #pragmas
- Normalize
  - E.g. loop-distribute a memset from the rest of a loop
- Apply platform-dependent transformations on recognized idioms
  - Replace initialization to 0 by memset (i.e. LoopIdiom)
  - Detect FFT -> Call fftw
  - ...
- Apply polyhedral reschedule (isl)
  - Feautrier
  - PLuTo
- Apply general transformations
  - Unroll for small loops
  - Vectorize with cost model
  - ...
- Use cost model to chose fastest variant
- Be conservative unless appropriate switch is used ("`clang -O42`")

LOOP OPTIMIZATION FRAMEWORK

Generate VPlan

- Determine the outermost loop(s) that has changed
- Generate runtime conditions from assumptions
  - noalias, overflow, integer ranges, alignment, etc.
- Preserve old loop nest for versioning
  - Special case: try to use for vectorization epilogue as well

- Generate LLVM-IR

```
for.body4:
  %indvars.iv = phi i64 [ 127, %for.cond1.preheader ], [ %indvars.iv.next, %for.body4 ]
  %1 = trunc i64 %indvars.iv to i32
  %conv = sitofp i32 %1 to double
  %div = fmul fast double %mul7, %conv
  %2 = tail call fast double @llvm.cos.f64(double %div)
  %mul8 = fmul fast double %2, %conv
  %arrayidx10 = getelementptr inbounds [128 x double]* @B, i64 0, i64 %indvars.iv24, i64 %indvars.iv
  store double %mul8, double* %arrayidx10, align 8, !tbaa !5
  %indvars.iv.next = add nsw i64 %indvars.iv, -1
  %cmp2 = icmp eq i64 %indvars.iv, 0
  br i1 %cmp2, label %for.cond.cleanup3, label %for.body4, !llvm.loop !9
```

- or -

- Generate VPlan

Vectorization Plan
Initial VPlan for VF={2,4},UF>=1

```
for.body4:
WIDEN-INDUCTION %indvars.iv = phi 0, %indvars.iv.next
WIDEN-INDUCTION
%indvars.iv = phi 0, %indvars.iv.next
%1 = trunc %indvars.iv
WIDEN
%conv = sitofp %1
%div = fmul %mul7, %conv
%2 = call %div, @llvm.sin.f64
%mul8 = fmul %2, %conv
CLONE %arrayidx10 = getelementptr @A, 0, %indvars.iv25, %indvars.iv
WIDEN store %mul8, %arrayidx10
```

```
if (rtc) {
  /* generated code */
} else {
  /* original code */
}
```

LOOP OPTIMIZATION FRAMEWORK

Not Yet Mentioned

- Dependency analysis:
  - Register dependency
  - Control dependency
- Import/Export of Loop Trees
- Online Autotuning
  - Compile to fat binary, with low base optimization
  - Sampling-profile-guided optimization
  - Gradually try riskier transformations in hot code, inline larger chunks
  - Call external library to generate code versions from DSLs
- Data-layout transformations
  - JIT kernels to use current data layout

Acknowledgments

- The LLVM community (including our many contributing vendors)
- ALCF, ANL, and DOE
- ALCF is supported by DOE/SC under contract DE-AC02-06CH11357

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.