Optimizing Indirections, or using abstractions without remorse

LLVMDev'18 — October 18, 2018 — San Jose, California, USA

Johannes Doerfert, Hal Finkel

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

Argonne NATIONAL LABORATORY

Acknowledgment

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.

Context & Motivation

Optimizations for *sequential* aspects

Optimizations for *parallel* aspects

a

b

Optimizations for *sequential* aspects

- Can reuse (improved) *existing* transformations

Optimizations for *parallel* aspects

a

b

Context — Optimizations For Parallel Programs

Optimizations for *sequential* aspects

- Can reuse (improved) *existing* transformations

Optimizations for *parallel* aspects

- New *explicit parallelism-aware transformations* (see IWOMP'18[a])

[a] *Compiler Optimizations For OpenMP*, J. Doerfert, H. Finkel, IWOMP 2018

[b]

Context — Optimizations For Parallel Programs

Optimizations for *sequential* aspects

- Can reuse (improved) *existing* transformations

⇒ Introduce *suitable abstractions and transformations* to bridge the indirection

Optimizations for *parallel* aspects

- New *explicit parallelism-aware transformations* (see IWOMP'18[a])

[a] *Compiler Optimizations For OpenMP*, J. Doerfert, H. Finkel, IWOMP 2018

[b]

Optimizations for *sequential* aspects

- Can reuse (improved) *existing* transformations

⇒ Introduce *suitable abstractions and transformations* to bridge the indirection

Optimizations for *parallel* aspects

- New *explicit parallelism-aware transformations* (see IWOMP'18[a])

⇒ Introduce a unifying abstraction layer (see EuroLLVM'18 Talk[b])

[a] *Compiler Optimizations For OpenMP*, J. Doerfert, H. Finkel, IWOMP 2018

[b] *A Parallel IR in Real Life: Optimizing OpenMP*, H. Finkel, J. Doerfert, X. Tian, G. Stelle, Euro-LLVM Meeting 2018

Optimizations for *sequential* aspects

- Can reuse (improved) *existing* transformations

⇒ Introduce *suitable abstractions and transformations* to bridge the indirection

Interested?
Contact me and come to our BoF!

Optimizations for *pa*

- New *explicit parallelism-aware transformations* (see IWOMP'18[a])

⇒ Introduce a unifying abstraction layer (see EuroLLVM'18 Talk[b])

[a] *Compiler Optimizations For OpenMP*, J. Doerfert, H. Finkel, IWOMP 2018
[b] *A Parallel IR in Real Life: Optimizing OpenMP*, H. Finkel, J. Doerfert, X. Tian, G. Stelle, Euro-LLVM Meeting 2018

Original Program

```
int y = 7;

for (i = 0; i < N; i++) {
  f(y, i);
}
g(y);
```

After Optimizations

Original Program

```
int y = 7;

for (i = 0; i < N; i++) {
  f(y, i);
}
g(y);
```

After Optimizations

```
for (i = 0; i < N; i++) {
  f(7, i);
}
g(7);
```

Original Program

```
int y = 7;
#pragma omp parallel for
for (i = 0; i < N; i++) {
  f(y, i);
}
g(y);
```

After Optimizations

Original Program

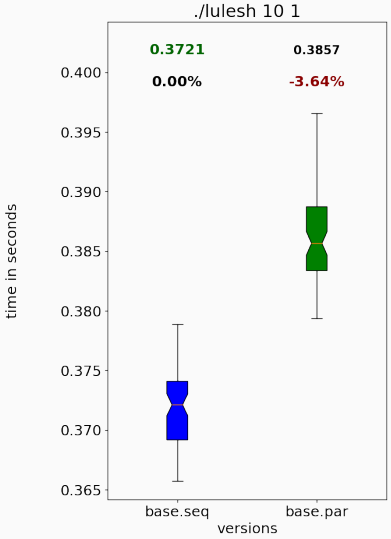
```
int y = 7;
#pragma omp parallel for
for (i = 0; i < N; i++) {
  f(y, i);
}
g(y);
```

After Optimizations

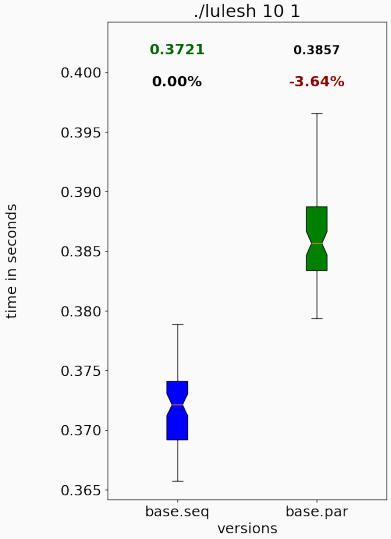
```
int y = 7;
#pragma omp parallel for
for (i = 0; i < N; i++) {
  f(y, i);
}
g(y);
```

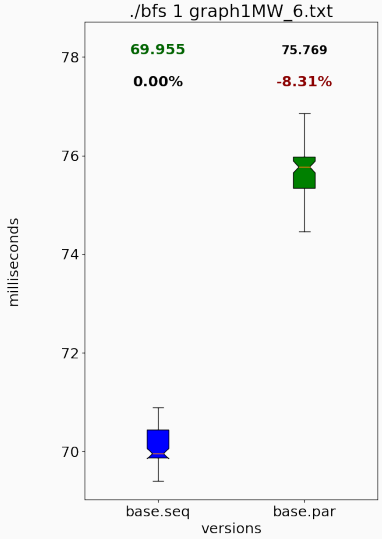
Why is this important?

Sequential Performance of Parallel Programs

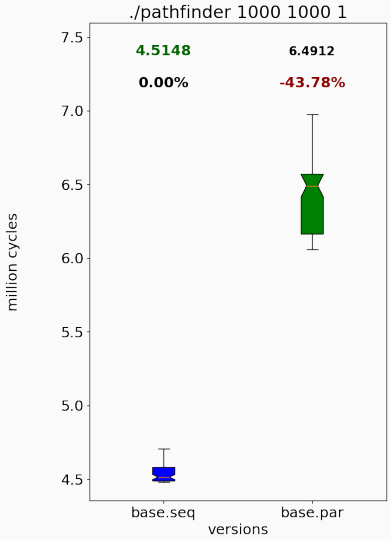


Sequential Performance of Parallel Programs

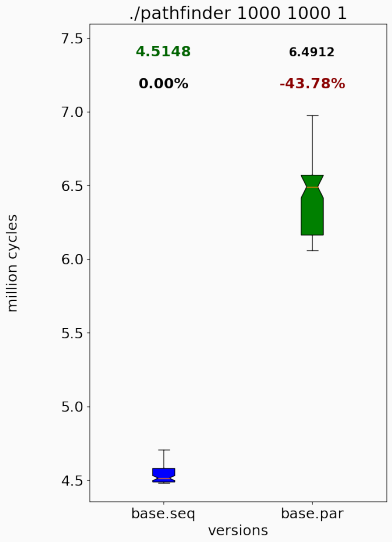


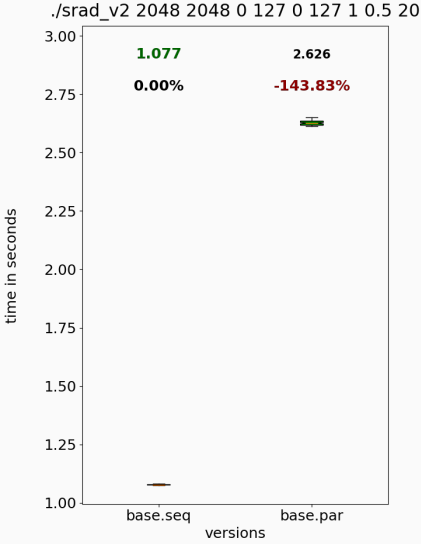


Sequential Performance of Parallel Programs



Sequential Performance of Parallel Programs





Early Outlining

OpenMP Input:

```
#pragma omp parallel for
for (int i = 0; i < N; i++)
  Out[i] = In[i] + In[i+N];
```

Early Outlining

OpenMP Input:

```
#pragma omp parallel for
for (int i = 0; i < N; i++)
  Out[i] = In[i] + In[i+N];
```

```
// Parallel region replaced by a runtime call.
omp_rt_parallel_for(0, N, &body_fn,   &N, &In, &Out);
```

Early Outlining

OpenMP Input:

```
#pragma omp parallel for
for (int i = 0; i < N; i++)
  Out[i] = In[i] + In[i+N];
```

```
// Parallel region replaced by a runtime call.
omp_rt_parallel_for(0, N, &body_fn,   &N, &In, &Out);


// Parallel region outlined in the front-end (clang)!
static void body_fn(int tid, int *N, float** In, float** Out) {
  int lb = omp_get_lb(tid), ub = omp_get_ub(tid);
  for (int i = lb; i < ub; i++)
    (*Out)[i] = (*In)[i] + (*In)[i + (*N)]
}
```

OpenMP Input:

```
#pragma omp parallel for
for (int i = 0; i < N; i++)
  Out[i] = In[i] + In[i+N];
```

```
// Parallel region replaced by a runtime call.
omp_rt_parallel_for(0, N, &body_fn,   &N, &In, &Out);


// Parallel region outlined in the front-end (clang)!
static void body_fn(int tid, int* N, float** In, float** Out) {
  int lb = omp_get_lb(tid), ub = omp_get_ub(tid);
  for (int i = lb; i < ub; i++)
    (*Out)[i] = (*In)[i] + (*In)[i + (*N)]
}
```

An Abstract Parallel IR

OpenMP Input:

```
#pragma omp parallel for
for (int i = 0; i < N; i++)
  Out[i] = In[i] + In[i+N];
```

```
// Parallel region replaced by an annotated loop
parfor (int i = 0; i < N; i++)
  body_fn(i, &N, &In, &Out);

// Parallel region outlined in the front-end (clang)!
static void body_fn(int  i , int* N, float** In, float** Out) {


    (*Out)[i] = (*In)[i] + (*In)[i + (*N)]
}
```

OpenMP Input:

```
#pragma omp parallel for
for (int i = 0; i < N; i++)
  Out[i] = In[i] + In[i+N];
```

```
// Parallel region replaced by a runtime call.
omp_rt_parallel_for(0, N, &body_fn,   &N, &In, &Out);


// Parallel region outlined in the front-end (clang)!
static void body_fn(int tid, int* N, float** In, float** Out) {
  int lb = omp_get_lb(tid), ub = omp_get_ub(tid);
  for (int i = lb; i < ub; i++)
    (*Out)[i] = (*In)[i] + (*In)[i + (*N)]
}
```

Early Outlining + Transitive Calls

OpenMP Input:

```
#pragma omp parallel for
for (int i = 0; i < N; i++)
  Out[i] = In[i] + In[i+N];
```

```
// Parallel region replaced by a runtime call.
omp_rt_parallel_for(0, N, &body_fn,   &N, &In, &Out);
// Model transitive call:  body_fn(?, &N, &In, &Out);

// Parallel region outlined in the front-end (clang)!
static void body_fn(int tid, int* N, float** In, float** Out) {
  int lb = omp_get_lb(tid), ub = omp_get_ub(tid);
  for (int i = lb; i < ub; i++)
    (*Out)[i] = (*In)[i] + (*In)[i + (*N)]
}
```

Early Outlining + Transitive Calls

OpenMP Input:

```
#pragma omp parallel for
for (int i = 0; i < N; i++)
  Out[i] = In[i] + In[i+N];
```

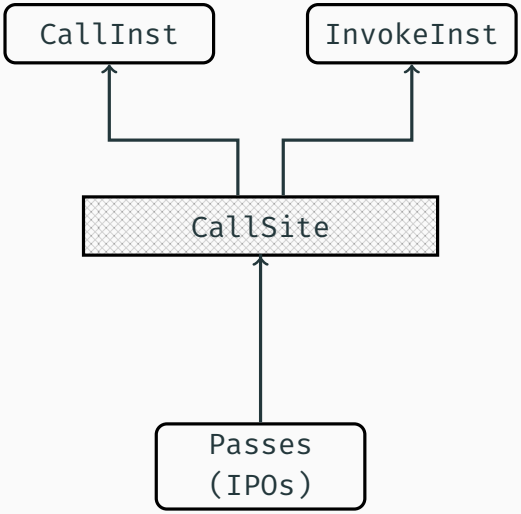
```
// Parallel region replaced by a runtime call.
omp_rt_parallel_for(0, N, &body_fn,   &N, &In, &Out);
// Model transitive call:  body_fn(?, &N, &In, &Out);

// Parallel
static void
  int lb =
  for (int
    (*Out)[
}
```

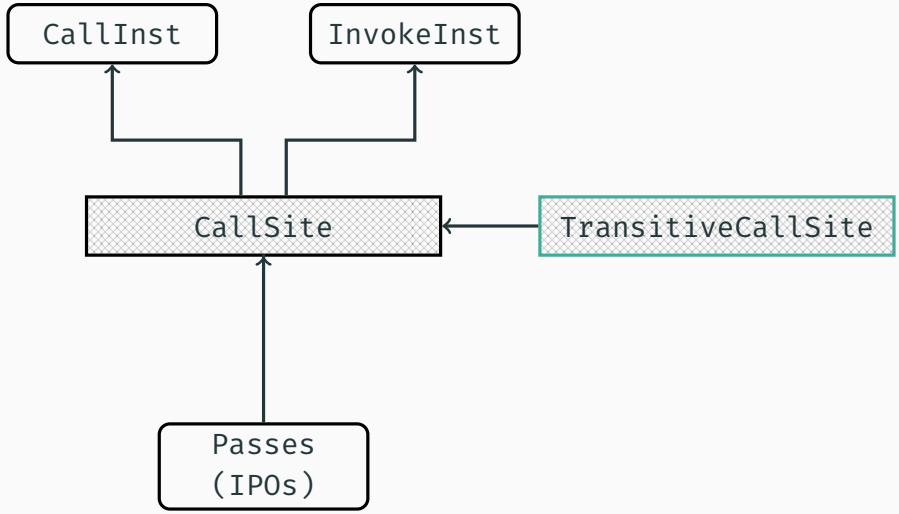
```
                                                  Out) {
```

- \+ *valid* and *executable* IR
- \+ *no unintended interactions*
- \+ *>1k function pointers arguments in LLVM-TS + SPEC*
- – *integration cost* per IPO

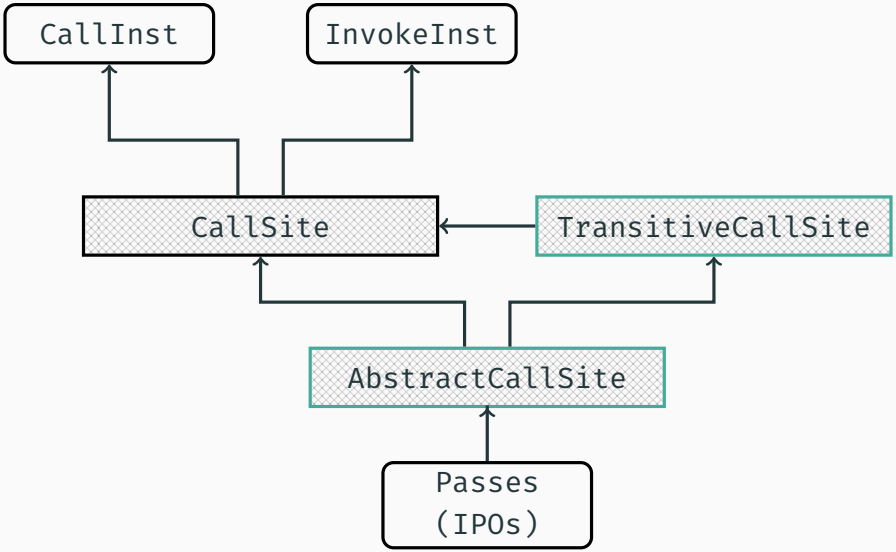
Call Abstraction in LLVM

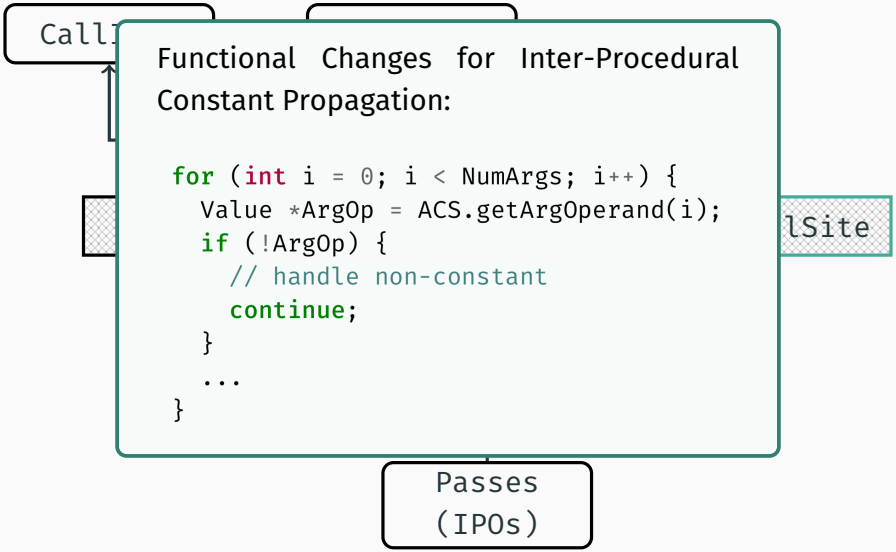


Call Abstraction in LLVM + Transitive Call Sites



Call Abstraction in LLVM + Transitive Call Sites

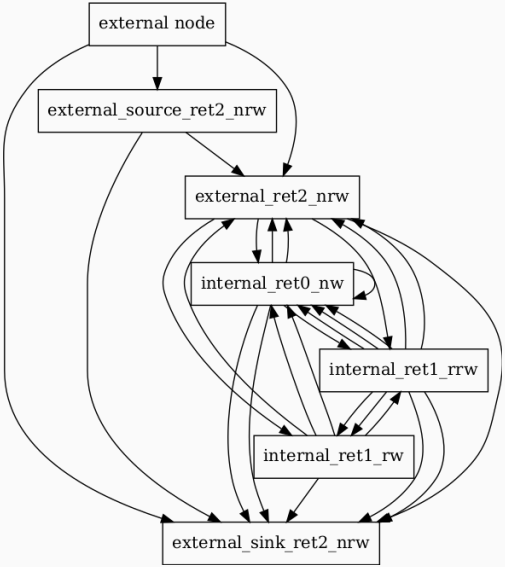


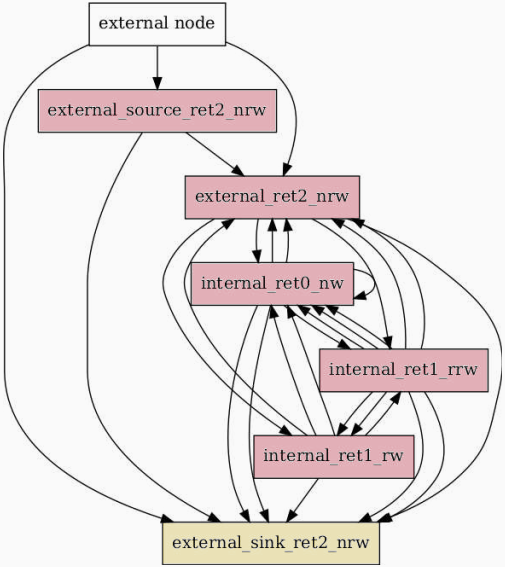

Call
lSite
Functional Changes for Inter-Procedural Constant Propagation:
for (int i = 0; i < NumArgs; i++) {
Value *ArgOp = ACS.getArgOperand(i);
if (!ArgOp) {
// handle non-constant
continue;
}
...
}
Passes
(IPOs)

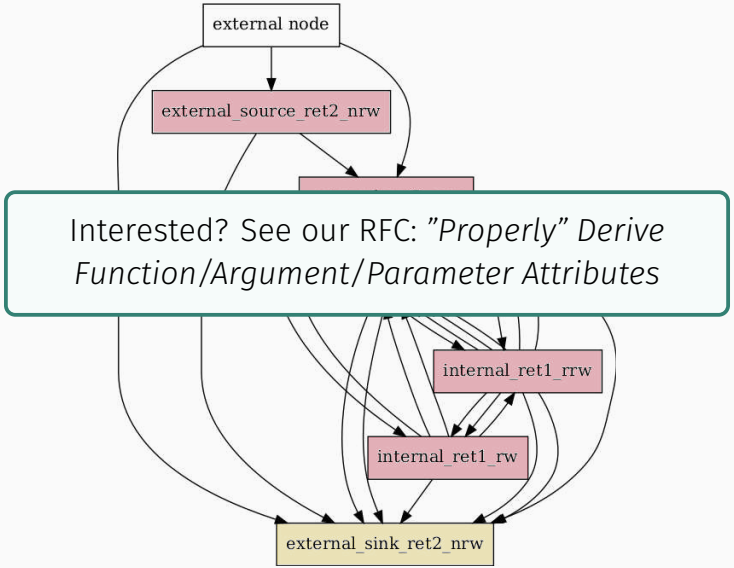
Inter-Procedural Optimization (IPO) in LLVM

```
static int* internal_ret1_rrw(int *r0, int *r1, int *w0);
static int* internal_ret0_nw(int *n0, int *w0);
static int* internal_ret1_rw(int *r0, int *w0);
int* external_source_ret2_nrw(int *n0, int *r0, int *w0);
int* external_sink_ret2_nrw(int *n0, int *r0, int *w0);
int* external_ret2_nrw(int *n0, int *r0, int *w0);
```

```
static int* internal_ret1_rrw(int *r0, int *r1, int *w0) {
  if (!*r0)
    return r1;
  internal_ret1_rw(r0, w0);
  *w0 = *r0 + *r1;
  internal_ret1_rw(r1, w0);
  internal_ret0_nw(r0, w0);
  internal_ret0_nw(w0, w0);
  external_ret2_nrw(r0, r1, w0);
  external_ret2_nrw(r1, r0, w0);
  external_sink_ret2_nrw(r0, r1, w0);
  external_sink_ret2_nrw(r1, r0, w0);
  return internal_ret0_nw(r1, w0);
}
```


external node
external_source_ret2_nrw
external_ret2_nrw
internal_ret0_nw
internal_ret1_rrw
internal_ret1_rw
external_sink_ret2_nrw


external node
external_source_ret2_nrw
external_ret2_nrw
internal_ret0_nw
internal_ret1_rrw
internal_ret1_rw
external_sink_ret2_nrw



Interested? See our RFC: *"Properly" Derive Function/Argument/Parameter Attributes*

```
static int foo(int a, int b) {
  return a + b; // 5?
}

int bar() {

  return foo(2, 3);
}
```

```
static int foo(int a, int b) {
  return a + b; // 5?
}

int bar() {

  return foo(2, 3);
}
```

```
struct Pair {
  int a, b;
};

static int foo(struct Pair p) {
  return p.a + p.b; // 5?
}

int bar() {
  struct Pair p = {2, 3};
  return foo(p);
}
```

```
static int foo(int a, int b) {
  return 5;
}

int bar() {

  return foo(2, 3);
}
```

```
struct Pair {
  int a, b;
};

static int foo(struct Pair p) {
  return 5;
}

int bar() {
  struct Pair p = {2, 3};
  return foo(p);
}
```

```c
struct Pair {
  int a, b;
};

static int foo(struct Pair *p) {
  return p->a + p->b; // 5?
}

int bar() {
  struct Pair p = {2, 3};
  return foo(&p);
}
```

```c
struct Tuple {
  int a, b, c, d;
};

static int foo(struct Tuple t) {
  return t.a + t.b + t.c + t.d; // 5?
}

int bar() {
  struct Tuple t = {2, 3, 0, 0};
  return foo(t);
}
```

```
struct Pair {
  int a, b;
};

static int foo(struct Pair *p) {
  return p->a + p->b;
}

int bar() {
  struct Pair p = {2, 3};
  return foo(&p);
}
```

```
struct Tuple {
  int a, b, c, d;
};

static int foo(struct Tuple t) {
  return t.a + t.b + t.c + t.d;
}

int bar() {
  struct Tuple t = {2, 3, 0, 0};
  return foo(t);
}
```

```
struct Pair {
  int a, b;
};

static int foo(s
  return p->a +
}

int bar() {
  struct Pair p = {2, 3};
  return foo(&p);
}
```

```
struct Tuple {
  int a, b, c, d;
};

ct Tuple t) {
+ t.c + t.d;

int bar() {
  struct Tuple t = {2, 3, 0, 0};
  return foo(t);
}
```

Why? Pipeline is less tuned and passes are conservative for IPO.

```
struct Tuple {
  int a, b, c, d, e, f, g, *h;
};
static
int f(struct Tuple *t) {
  return t->a+t->c+t->e+t->g;
}

int bar(struct Tuple *t) {
  t->a = 3; t->e = 7; /* ... */
  f(t);
  // ... t->h does escape in f!
}
```

```
struct Tuple {
  int a, b, c, d, e, f, g, *h;
};
static
int f(struct Tuple *t) {
  return t->a+t->c+t->e+t->g;
}

int bar(struct Tuple *t) {
  t->a = 3; t->e = 7; /* ... */
  f(t);
  // ... t->h does escape in f!
}
```

```
struct Tuple {
  int a, b, c, d, e, f, g, *h;
};
static
int f(int a, int c, int e, int g) {
  return a + c + e + g;
}

int bar(struct Tuple *t) {
  t->a = 3; t->e = 7; /* ... */
  f(3, t->c, 7, t->g);
  // ... t->h does not escape in f!
}
```

```
struct Tuple {
  int a, b, c, d, e, f, g, *h;
};
static
int f(struct Tuple *t) {
  return t->a+t->c+t->e+t->g;
}

int bar(struct Tuple *t) {
  t->a = 3; t->e = 7; /* ... */
  f(t);
  // ... t->h does escape in f!
}
```

```
struct Tuple {
  int a, b, c, d, e, f, g, *h;
};
static
int f(int c, int g) {
  return 3 + c + 3 + g;
}

int bar(struct Tuple *t) {
  t->a = 3; t->e = 7; /* ... */
  f(t->c, t->g);
  // ... t->h does escape in f!
}
```

```
struct Tuple {
  int a, b, c, d, e, f, g, *h;
};
static
int f(struct Tuple *t) {
  return t->a+t->c+t->e+t->g;
}

int bar(struct Tuple *t) {
  t->a = 3; t->e = 7; /* ... */
  f(t);
  // ... t->h does escape in f!
}
```

```
struct Tuple {
  int a, b, c, d, e, f, g, *h;
};
static
int f(struct Tuple *t) {
  return 3 + t->c + 7 + t->g;
}

int bar(struct Tuple *t) {
  t->a = 3; t->e = 7; /* ... */
  f(t);
  // ... t->h does escape in f!
}
```

```
struct Tuple {
  int a, b, c, d, e, f, g, *h;
};
static
int f(struct Tup
  return t->a+t-
}

int bar(struct T
  t->a = 3; t->e = 7; /* ... */
  f(t);
  // ... t->h does escape in f!
}
```

```
struct Tuple {
  int a, b, c, d, e, f, g, *h;
};
                          *t) {
                        7 + t->g;

                        e *t) {
  t->a = 3; t->e = 7; /* ... */
  f(t);
  // ... t->h does escape in f!
}
```

Aggressively unpack object arguments early and condense arguments late as an alternative/substitution for inlining.

IPO — Additional Proposals/Prototypes

- track values of fields across function calls, e.g., closure initialization [prototype]
- determine performance impact of missing static information [ongoing]
- export attributes for libraries, e.g., add `__attribute__((const))` [planned]

IPO — Additional Proposals/Prototypes

- track values of fields across function calls, e.g., closure initialization [prototype]
- determine performance impact of missing static information [ongoing]
- export attributes for libraries, e.g., add `__attribute__((const))` [planned]

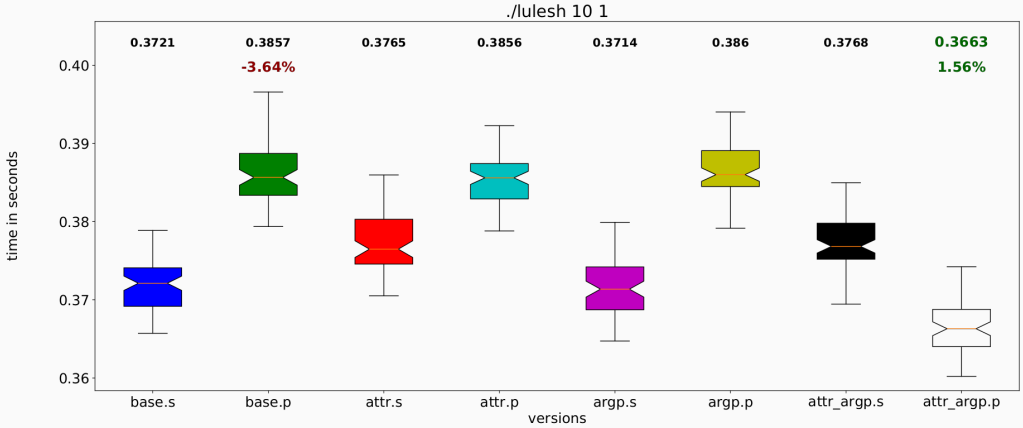
Interested? Contact me!

Evaluation

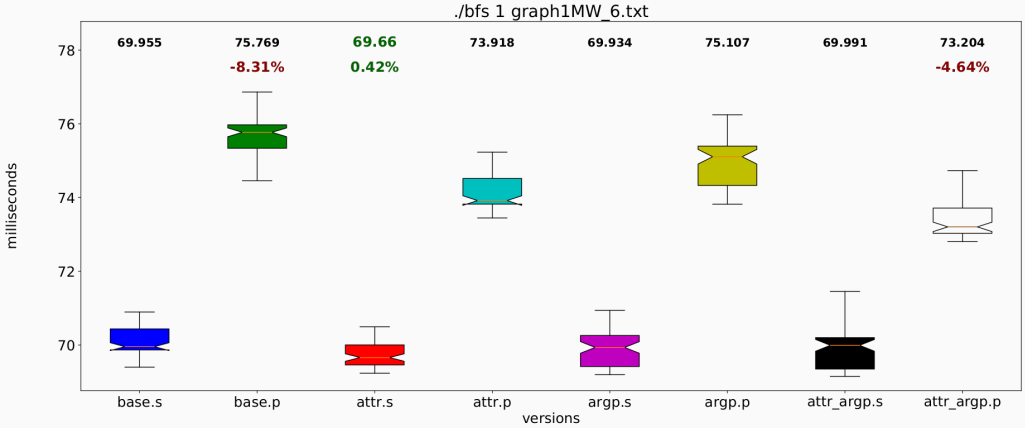
OpenMP Optimizations

Version	Description	Opt.
base	plain “-O3”, thus no parallel optimizations	
attr	attribute propagation through attr. deduction (IPO)	I
argp	variable privatization through arg. promotion (IPO)	II
n/a	constant propagation (IPO)	

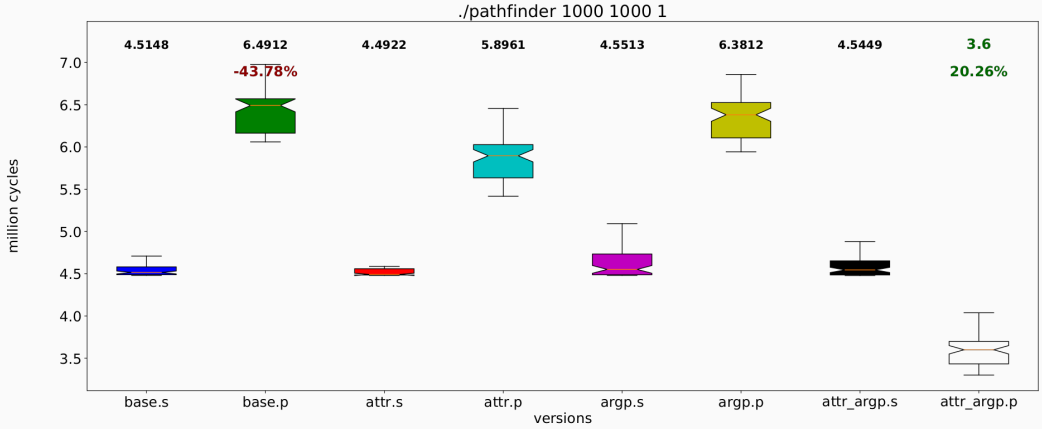
OpenMP Optimizations — Performance Results



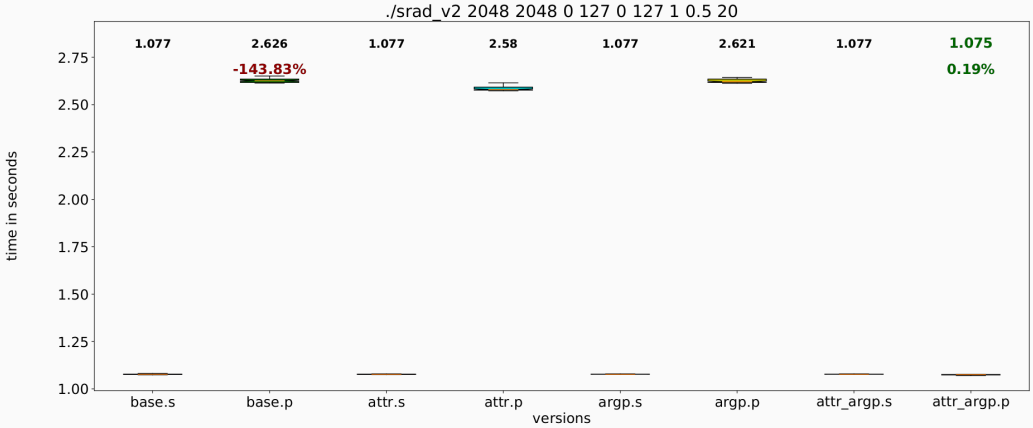
OpenMP Optimizations — Performance Results



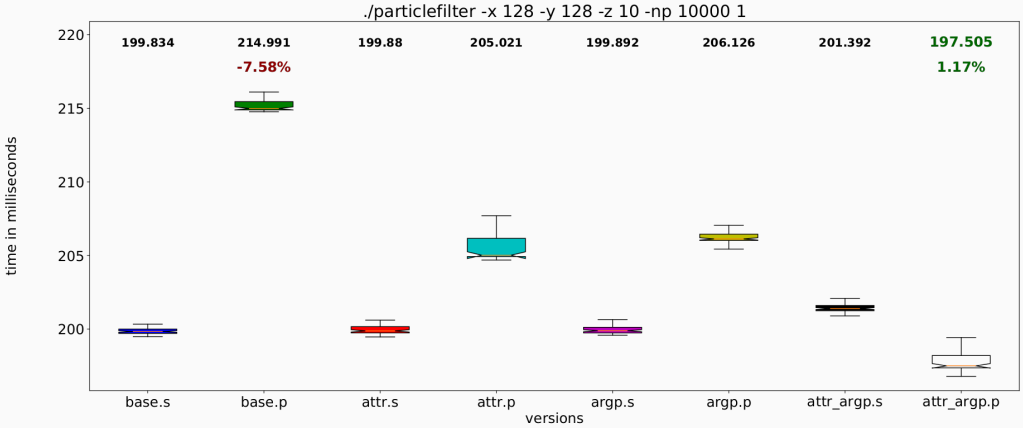
OpenMP Optimizations — Performance Results



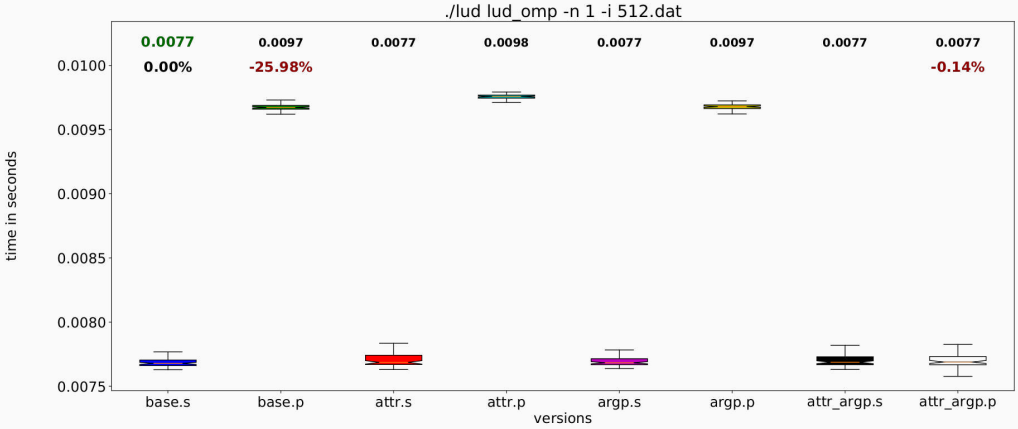
OpenMP Optimizations — Performance Results



OpenMP Optimizations — Performance Results



OpenMP Optimizations — Performance Results

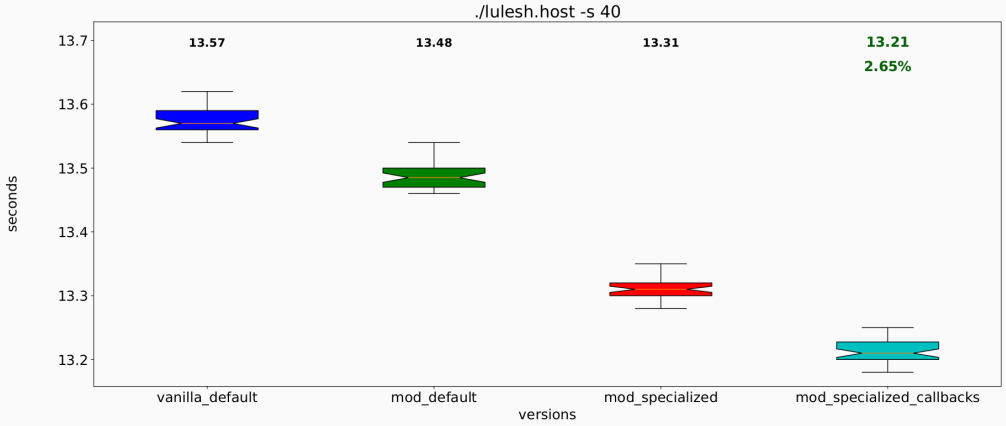


Array Constant Propagation Example

```
double gamma[4][8];
gamma[0][0] = 1;
// ... and so on till ...
gamma[3][7] = -1;

Kokkos::parallel_for(
  "CalcFBHourglassForceForElems A",
  numElem, KOKKOS_LAMBDA(const int &i2) {
  // Use gamma[0][0] ... gamme[3][7]
}
```

Array Constant Propagation Performance

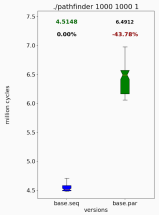


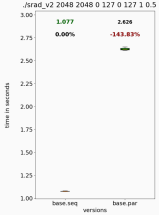
Conclusion

Conclusion

Conclusion

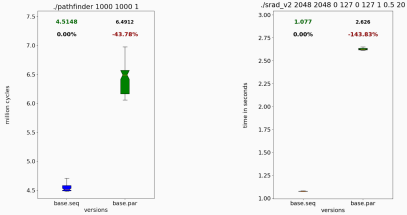
Sequential Performance of Parallel Programs



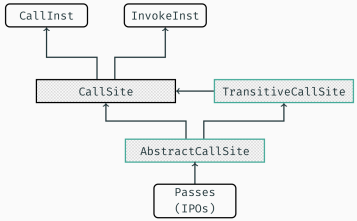


Conclusion

Sequential Performance of Parallel Programs

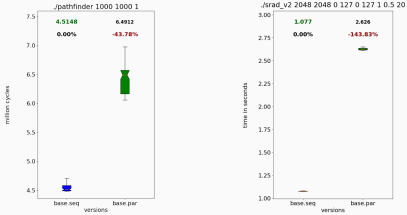


Call Abstraction in LLVM + Transitive Call Sites

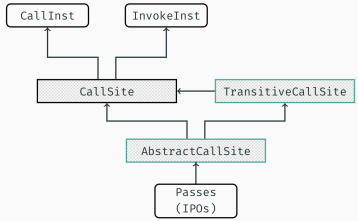


Conclusion

Sequential Performance of Parallel Programs

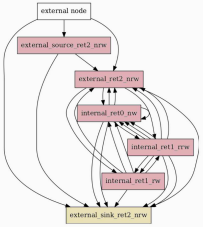


Call Abstraction in LLVM + Transitive Call Sites



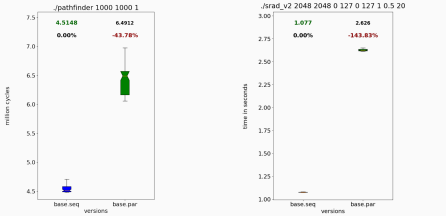
IPO — Attribute Inference

-O3 -disable-inlining

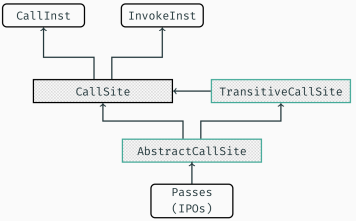


Conclusion

Sequential Performance of Parallel Programs

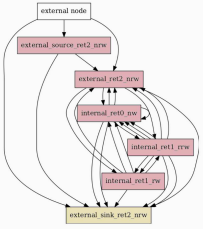


Call Abstraction in LLVM + Transitive Call Sites



IPO — Attribute Inference

-O3 -disable-inlining



OpenMP Optimizations — Performance Results