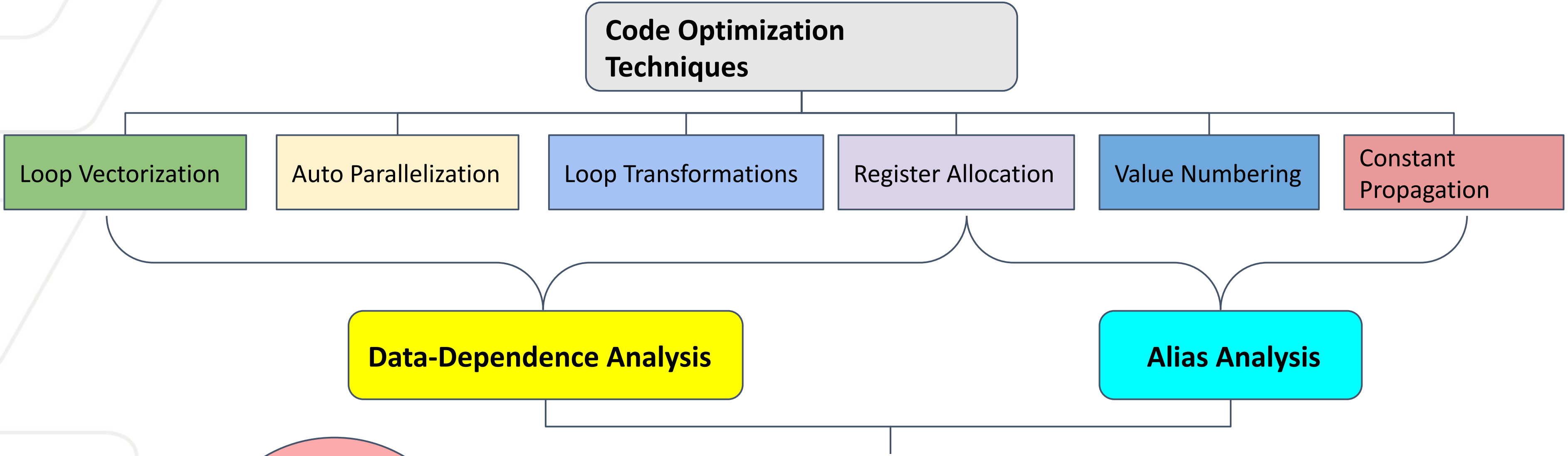
VICO: Demand-Driven Verification for Improving Compiler Optimizations

Sharjeel Khan Bodhisatwa Chatterjee Santosh Pande

Georgia Tech

Motivation: Traditional Compiler Analysis suffer from Imprecision



Conservative (Safe) Approximations

Hinders optimization opportunities

- ❌ Inability to check infeasible program paths
- ❌ Inability to symbolically propagate and evaluate expressions
- ❌ inability to verify statically unknown properties *inter-procedurally*

Georgia Tech

Example: Liebmann's Method with generalized boundary conditions

```
int getK (int par) {

if (par % 2)
  return 2*(par + 1);
else
 return 2*par;
}
```

Possible implementation of boundary offset values

k1 and k2 initialized by external function calls

```
void liebmann2D (/*arguments*/) {

int k1 = getK(N), k2 = getK(N);
for (t = 0; t <= M; t++)
 for (i = 1; i <= N; i++)
  for (j = 1; j <= N; j++)
        A[i][j] = (A [i - k1] [j - k1] + A[i - k1][j] + A[i][j]
             + A[i - k1][j + k2] + A[i][j - k1] + A[i][j + k2]
             + A[i + k2][j - k1] + A[i + k2][j]
             + A[i + k2][j + k2])/c;
}
```

Interprocedural whole program flow analysis unable to prove the (k1,k2 > N) invariant

⇧

Proving (k1,k2 > N) breaks most dependencies

⇧

$0 \leq k1, k2 \leq N$ (Dependence Equation)
Compiler assumes all possible dependencies

Demand-driven verification based solution

Proving Optimization Constraints

↓

Demand-Driven Verification

⇩

✅ Proves only those properties that are related to optimization instance at hand

✅ Has the ability to pick properties that can break maximum constraints

Use of Software Verification in Compilers

↓

Verification for Compiler Optimizations

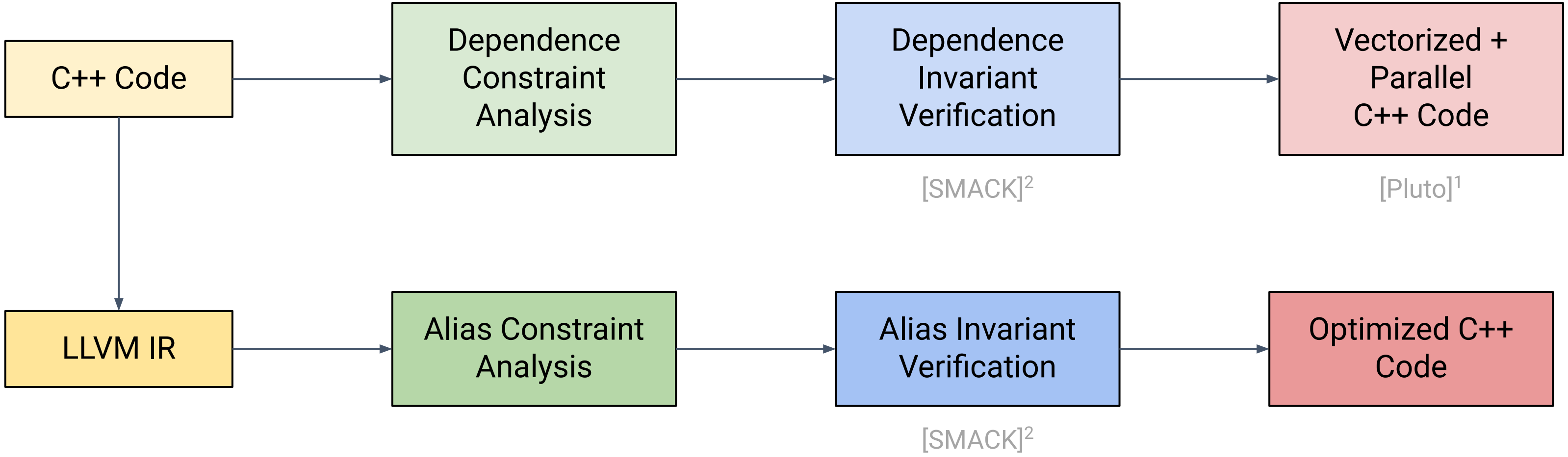
To the best of our knowledge, this line of work has not been tackled previously

⇩

⏩ Use of software verification in a demand-driven manner to boost compiler optimizations

⏩ Focus is on finding out the bottlenecks for compiler analysis, formulate the necessary invariants and then verify them - demand driven

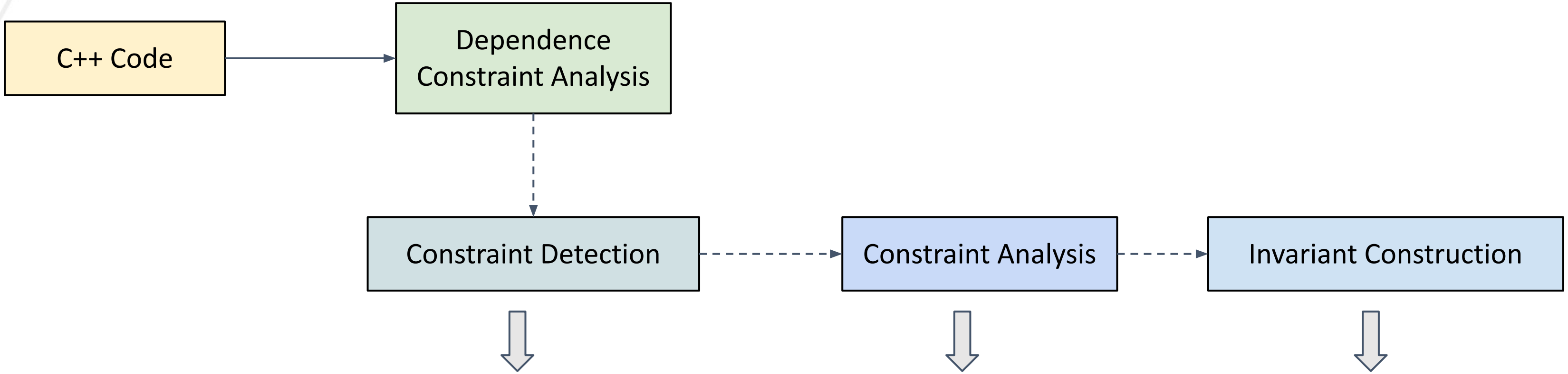
VICO: Demand-Driven Verification for Improving Compiler Optimization



1. https://github.com/bondhugula/pluto 2. https://github.com/smackers/smack

Georgia Tech

VICO: Demand-Driven Verification for Improving Compiler Optimization



- PLuTo[1] dependence logs
- Detecting *absolute* & *derived* constraints

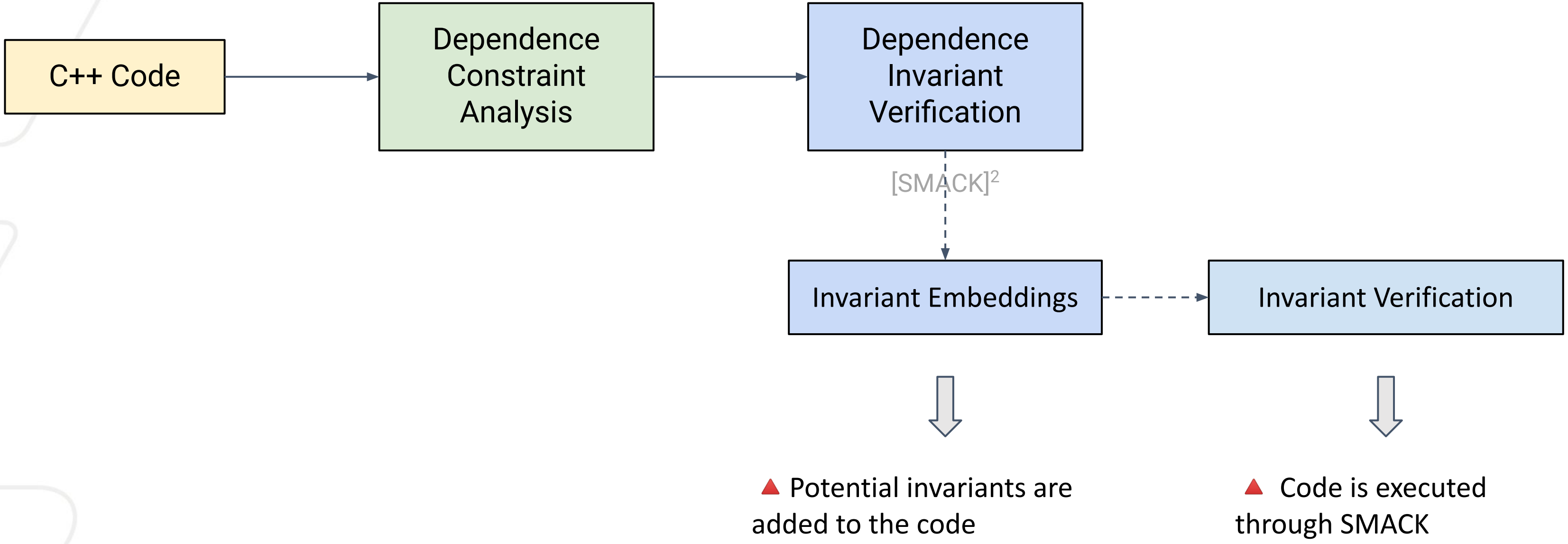
- Constraint transformation
- Constraint selection

- Operator flipping

1. https://github.com/bondhugula/pluto

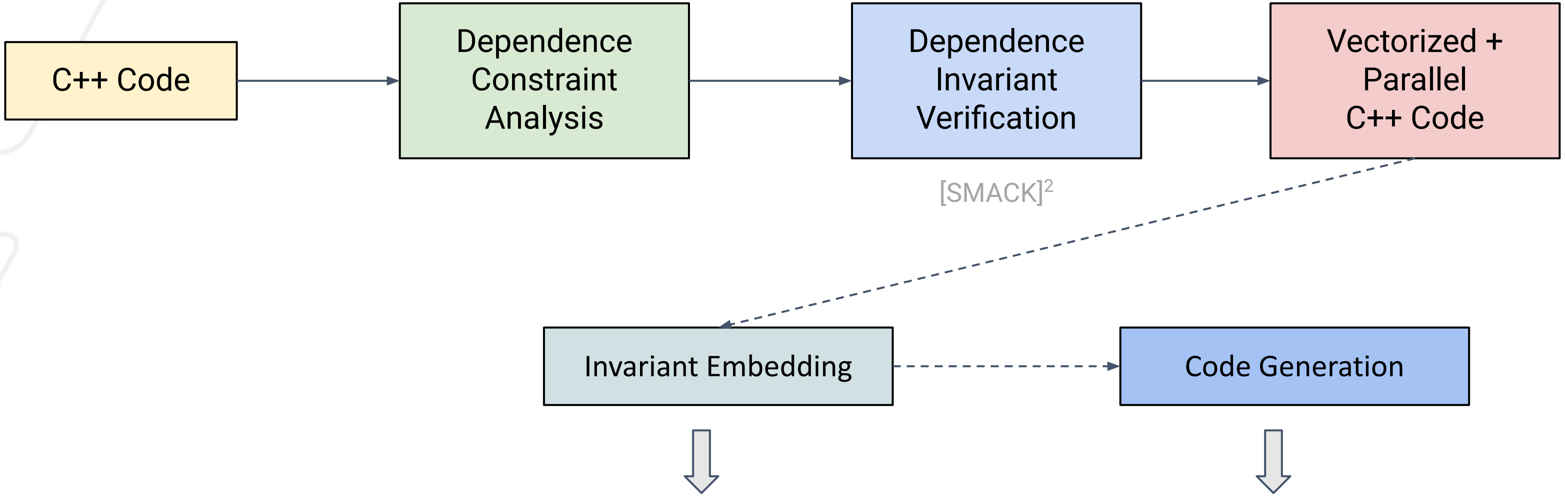
Georgia Tech

VICO: Demand-Driven Verification for Improving Compiler Optimization



1. https://github.com/bondhugula/pluto 2. https://github.com/smackers/smack

VICO: Demand-Driven Verification for Improving Compiler Optimization



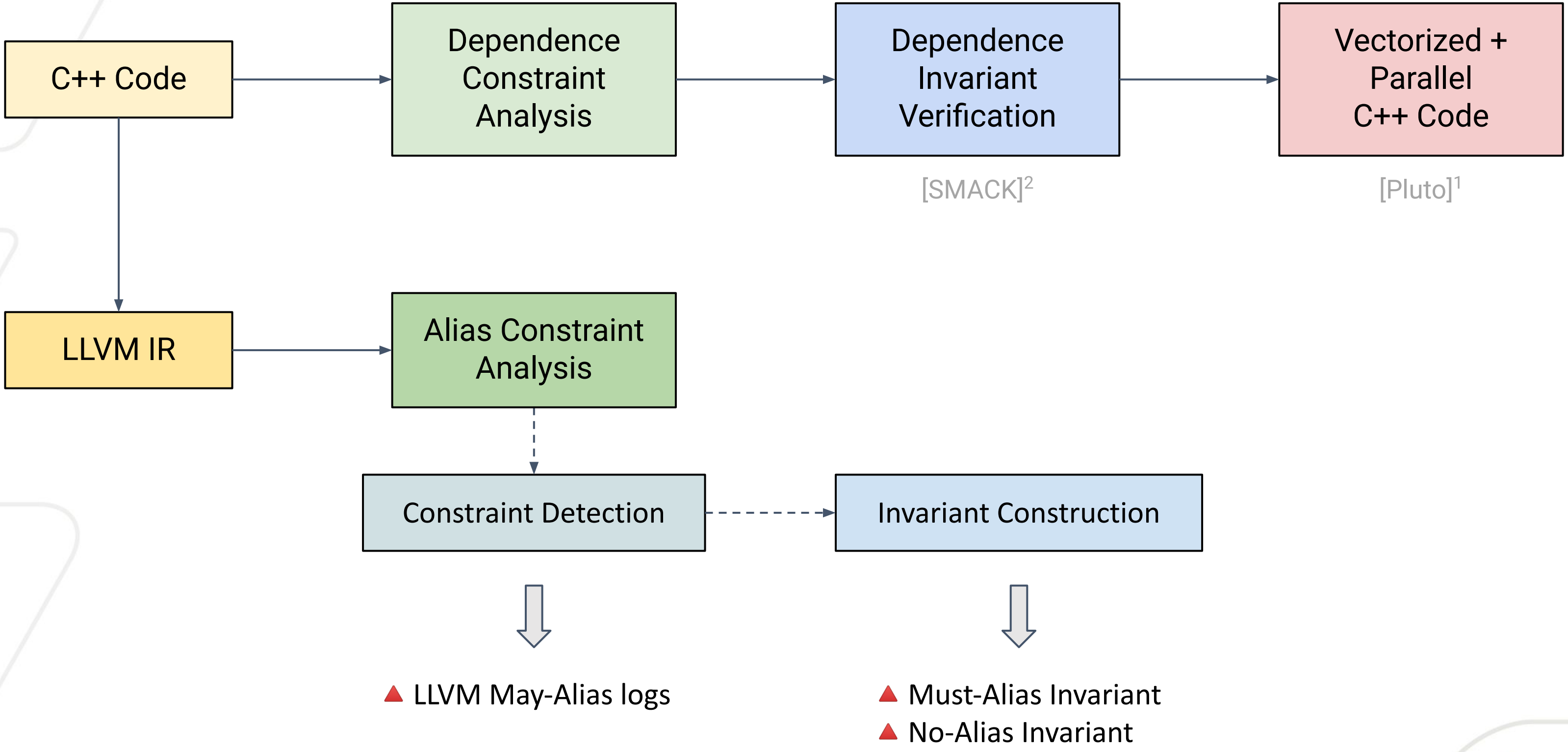
- Choose invariants that break maximum number of dependencies
- Add invariants in the original code

- PLuTo[1] generates optimized code from the embedded code

1. https://github.com/bondhugula/pluto 2. https://github.com/smackers/smack

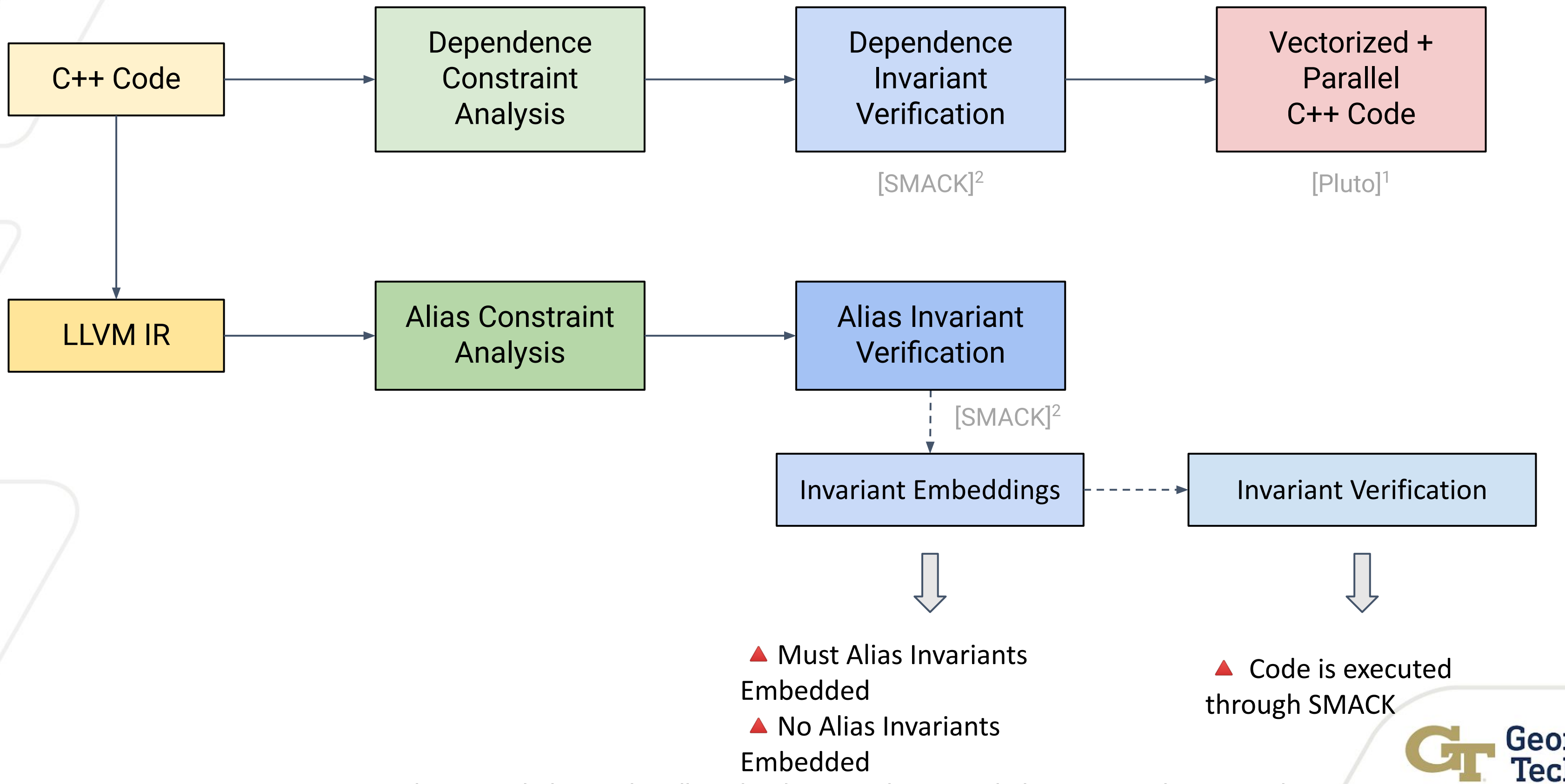
Georgia Tech

VICO: Demand-Driven Verification for Improving Compiler Optimization



1. https://github.com/bondhugula/pluto 2. https://github.com/smackers/smack

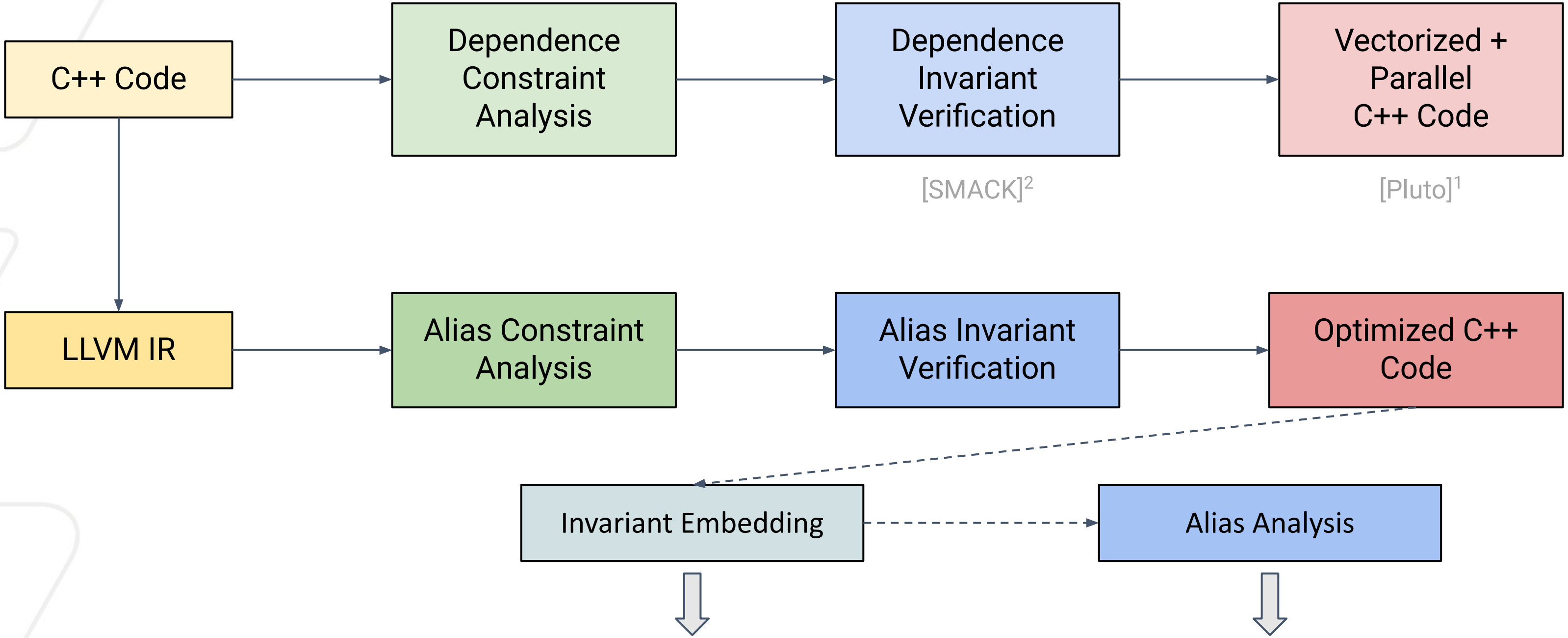
VICO: Demand-Driven Verification for Improving Compiler Optimization



1. https://github.com/bondhugula/pluto 2. https://github.com/smackers/smack

Georgia Tech

VICO: Demand-Driven Verification for Improving Compiler Optimization



- Add invariants in the original code

- Verified invariants inserted as SMACK asserts
- Alias pass finds saves the invariants in a map
- Allows chaining for other cases

Georgia Tech

1. https://github.com/bondhugula/pluto 2. https://github.com/smackers/smack

Summary of Results

- Improving precision of dependence analysis by 45% in real-world cases
  - Better parallelization techniques in over 75 loops
  - Average speed-up of 14.7x on Apple M1 Pro
  - Average speed-up of 6.07x on Intel Xeon E5-2660
  - Took a total time of more than 5 hours to verify all dependence constraints
- Improving precision of alias analysis
  - Average code size reduction by 1.621% with up to 4.1% in real-world applications
  - Average speed-up of 2.2% on Intel Xeon E5-2660
  - Average improvement in load/store instructions of 4.227% with up to 7.08% in real-world applications
  - Took a total time of more than 6 hours to verify the 93 alias constraints

Conclusion

- VICO: A Demand-Driven Verification Framework for improving Compiler Optimizations
  - Improves both dependence analysis and alias analysis
  - To the best of our knowledge, this is the first paper that leveraged verification to **enhance** compiler optimizations (*Note that this is very different problem than verifying compiler optimizations*).

- Future work
  - Target other optimizations, more complex invariants
  - Improve LLVM and Smack interactions

Georgia Tech

Backup Slides

Solution: Need for a demand-driven verification based solution

Proving Optimization Constraints

Whole Program Interprocedural Analysis

❌ Exponential number of program paths

❌ Edge based, context-insensitive, flow Insensitive approximations

❌ Whole program = unnecessary propagation, slow

❌ Not supported by most production compilers

Whole Program Verification

✅ Leverages pruning techniques to counter exponential path growth

❌ Proves all possible properties unrelated to optimization

❌ Doesn't have a starting point for choosing properties

Demand-Driven Verification

✅ Proves only those properties that are related to optimization instance at hand

✅ Has the ability to pick properties that can break maximum constraints

Georgia Tech

Vectorized + Parallelized C++ Code

Invariant Embedding → Code Generation

```
void liebmann2D (/*arguments*/) {

int k1 = getK(N), k2 = getK(N);
#pragma scop
if (k2 > N) {
for (t = 0; t <= M; t++)
 for (i = 1; i <= N - 2; i++)
  for (j = 1; j <= N - 2; j++)
          A[i][j] = (A [i - k1] [j - k1] + A[i - k1][j] + A[i][j]
               + A[i - k1][j + k2] + A[i][j - k1] + A[i][j + k2]
               + A[i + k2][j - k1] + A[i + k2][j]
               + A[i + k2][j + k2])/c;
 }
}
#pragma endscop
```

Embedded C/C++ Code

```
void liebmann2D (/*arguments*/) {
int k1 = getK(N), k2 = getK(N);
for (t = 0; t <= 2*M+N; t++) {
     lbp = max(ceild(t+1, 2),t-M+1);
     ubp = min(floord(t+N, 0),t);
     #pragma omp parallel for private(lbv,ubv,j)
     for (i = lbp; i <= ubp; i++)
       for (j = t + 1; j <= t + N; j++)
         A[(-t+2*i)][(-t+j)] = (A[(-t+2*i)-1][(-t+j)-1] + A[(-t+2*i)-1][(-t+j)]
                              + A[(-t+2*i)-1][(-t+j)+1]
                              + A[(-t+2*i)][(-t+j)-1]
                              + A[(-t+2*i)][(-t+j)] + A[(-t+2*i)][(-t+j)+1]
                              + A[(-t+2*t2)+1][(-t+j)-1]
                              + A[(-t+2*i)+1][(-t+j)]
                              + A[(-t+2*i)+1][(-t+j)+1])/c;
}
```

Parallelized C/C++ Code

Optimized C++ Code

Invariant Embedding → Alias Analysis

```
int main (/*arguments*/) {
/* function body definitions */
int temp = getk(30);
if(temp >= 30)
   p = &l;
else if(temp >= 10 && temp < 20)
   p = &i;
else if(temp >= 0 && temp < 10)
   p = &j;
else
   p = &k;

for(i = 0; i < n; i +=1){
   assert(p = &l); assert(p != &k);
  assert(p != &j); assert(p !=&i);
   for(j = 0; j < n; j +=1) {
      for(k = 0; k < n; k +=1) {
         *p = *p + 1;
         A[i][j][k] = B[i][j][k] + 11;
      }}}
/* More Code */}
```

Original C/C++ Code with a verified Invariant

```
define dso_local i32 @main(i32 %0, i8** %1) #2 !dbg !356 {
50:                                    ; preds = %49
  %51 = icmp ne i32* %3, %6, !dbg !430, !verifier.code !344
  br i1 %51, label %53, label %52, !dbg !433, !verifier.code !344
52:                                    ; preds = %50
  call void @__VERIFIER_assert(i32 0), !dbg !430, !verifier.code !428
  br label %53, !dbg !430, !verifier.code !344
56:                                    ; preds = %55
  %57 = icmp ne i32* %3, %5, !dbg !435, !verifier.code !344
  br i1 %57, label %59, label %58, !dbg !438, !verifier.code !344
58:                                    ; preds = %56
  call void @__VERIFIER_assert(i32 0), !dbg !435, !verifier.code !428
  br label %59, !dbg !435, !verifier.code !344
62:                                    ; preds = %61
  %63 = icmp ne i32* %3, %4, !dbg !440, !verifier.code !344
  br i1 %63, label %65, label %64, !dbg !443, !verifier.code !344
64:                                    ; preds = %62
  call void @__VERIFIER_assert(i32 0), !dbg !440, !verifier.code !428
  br label %65, !dbg !440, !verifier.code !344
}
```

LLVM IR representation

```
%3 ≠ null  (p = &l)
%3 ≠ %6  (p != &k)
%3 ≠ %5  (p != &j)
%3 ≠ %4  (p != &i)
```

Our Alias Analysis saving the invariants

↓

LLVM's Alias Analysis

Georgia Tech