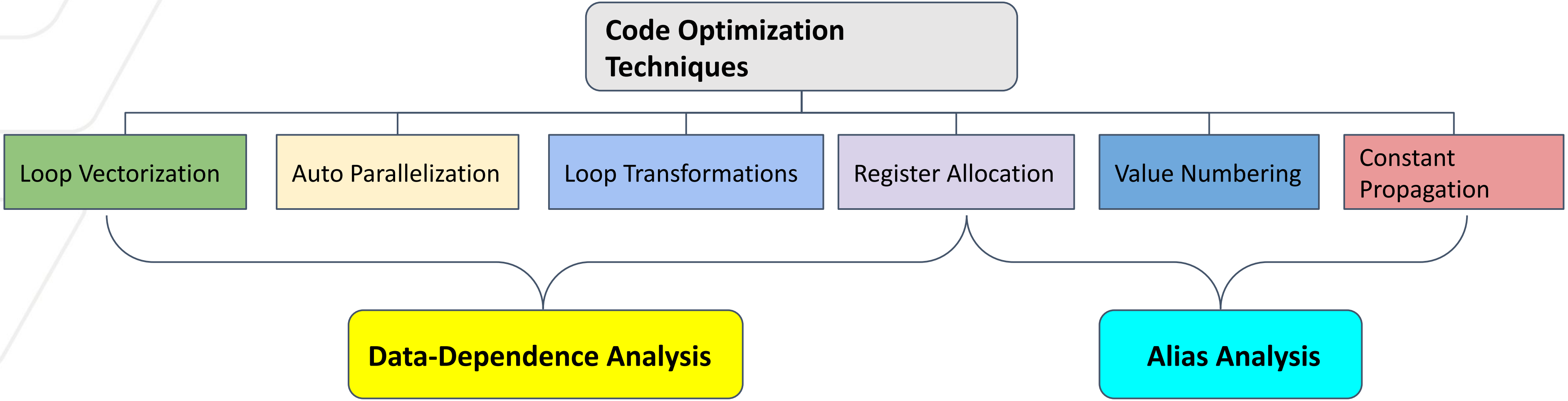
VICO: Demand-Driven Verification for Improving Compiler Optimizations

Sharjeel Khan Bodhisatwa Chatterjee Santosh Pande

Georgia Tech

Motivation: Traditional Compiler Analysis suffer from Imprecision



- ⏩ **Dependence Analysis** & **Alias Analysis** form the backbone of many important code optimization techniques
- ⏩ Goal of these analyses is to yield optimized end code, while keeping compilation time low

Georgia Tech

Motivation: Traditional Compiler Analysis suffer from Imprecision

Code Optimization Techniques

- Loop Vectorization
- Auto Parallelization
- Loop Transformations
- Register Allocation
- Value Numbering
- Constant Propagation

Data-Dependence Analysis

Alias Analysis

Hinders optimization opportunities

Conservative (Safe) Approximations

❌ Inability to check infeasible program paths

❌ Inability to symbolically propagate and evaluate expressions

❌ inability to verify statically unknown properties *inter-procedurally*

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 ≤ k1 , k2 ≤ N (Dependence Equation)
Compiler assumes all possible dependencies

Example: 505.mcf_r (SPEC 2017)

```
void marc_arcs(/*arguments*/) {
/* function body definitions */
while(global_new < *new_arcs && global_new < max_new_arcs) {
   if (values[0] < new_arcs_array[0])
      arc = *positions[0];
   else
      ..
   for (i = 1; i < num_threads; i++) {
      if ((values[i] < new_arcs_array[i]) &&
         ((arc_compare(positions[i], &arc) < 0) ||
          !arc)) {
            arc = *positions[i];
         }}
      global_new++;
   }
}
```

arc and *positions[0:num_threads] point to the same locations

⇩

LLVM considers these two as May-Aliases

⇩

arc and *positions[0:num_threads] should be considered as Must-Aliases

⇧

LLVM considers pointers which begin at same location and points to the same overlapping area as Must-Alias

Adverse effect on value numbering and register allocation

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

Key Insight: "Verification can boost Compiler Optimizations"

Use of Software Verification in Compilers

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

Verified Compiler Optimizations

- ⏩ Goal is to check the legality of various compiler optimizations
- ⏩ Focus is on checking the correctness of generated code i.e if the optimization preserves program semantics

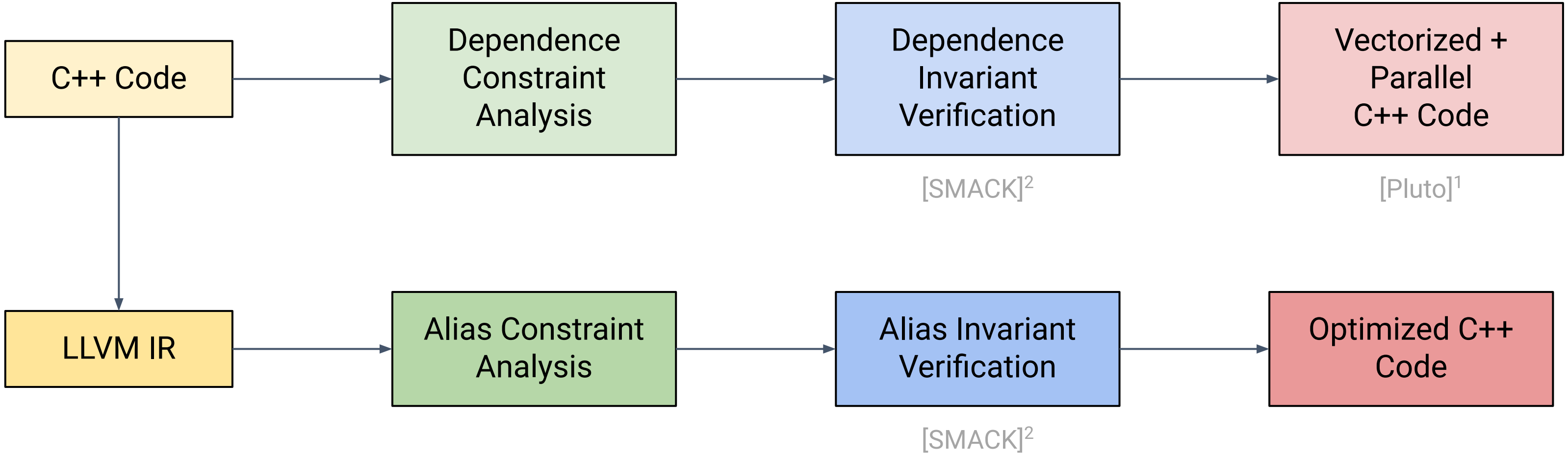
Formally Verified Compilers

- ⏩ Goal is to avoid miscompilation using mathematical reasoning
- ⏩ Focus is on proving the correctness of generated IR code

Verification for Compiler Optimizations

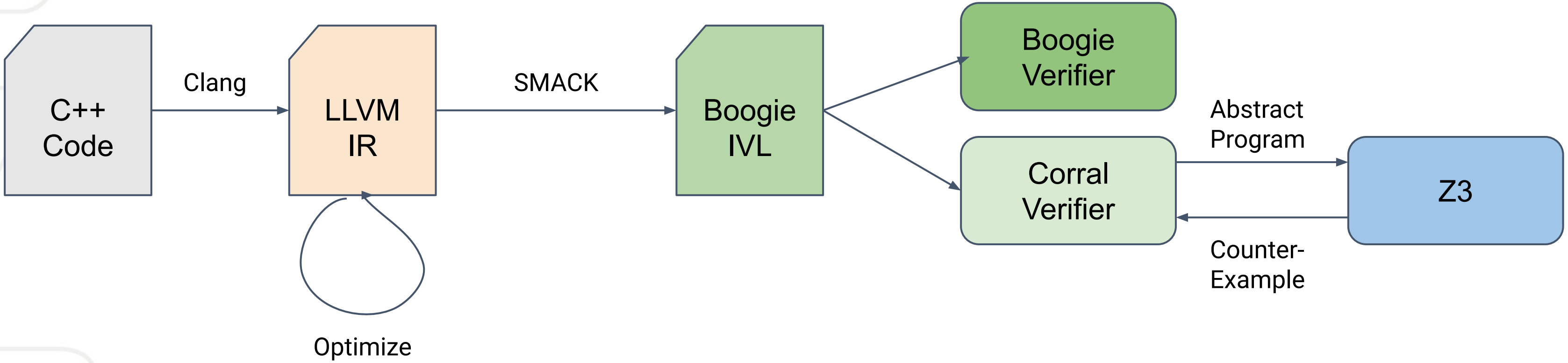
- ⏩ Use of software verification in a demand-driven manner to boost compiler optimizations
- ⏩ Focus is on finding out the bottlenecks for compiler analysis, formulate the necessary invariants and then verify them - demand driven

VICO: Demand-Driven Verification for Improving Compiler Optimization



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

Smack Verifier Toolchain[2]



A Verification Framework that uses LLVM optimizations and converts the IR into Boogie IVL to prove properties about the code

2. https://github.com/smackers/smack

Smack Example

```
int main () {
  int x = 0, i = 0;
  for (i  = 0; i < M; ++i) {
    x = i;
  }
  assert (x == M - 1);
  return 0;
}
```

Source Code

```
define dso_local i32 @main() #0 !dbg !34 {
      store i32 0, i32* @x, align 4, !dbg !40, !verifier.code !39
      br label %1, !dbg !41, !verifier.code !39
1:    preds = %4, %0
      %0 = phi i32 [ 0, %0 ], [ %5, %4 ], !dbg !43, !verifier.code !39
      %2 = icmp slt i32 %.0, 10, !dbg !44, !verifier.code !39
      br i1 %2, label %3, label %7, !dbg !46, !verifier.code !39
3:    preds = %1
      store i32 %.0, i32* @x, align 4, !dbg !47, !verifier.code !39
      br label %4, !dbg !49, !verifier.code !39
4:    preds = %3
      %5 = add nsw i32 %.0, 1, !dbg !50, !verifier.code !39
      br label %1, !dbg !51, !llvm.loop !52, !verifier.code !39
7:    preds = %1
      %8 = load i32, i32* @x, align 4, !dbg !56, !verifier.code !39
      %9 = icmp eq i32 %8, 9, !dbg !56, !verifier.code !39
      br i1 %9, label %12, label %10, !dbg !59, !verifier.code !39
10:   preds = %7
      call void @__VERIFIER_assert(i32 0), !dbg !56, !verifier.code
!60
      br label %12, !dbg !56, !verifier.code !39
12:   preds = %7, %10
      ret i32 0, !dbg !61, !verifier.code !39
}
```

LLVM IR

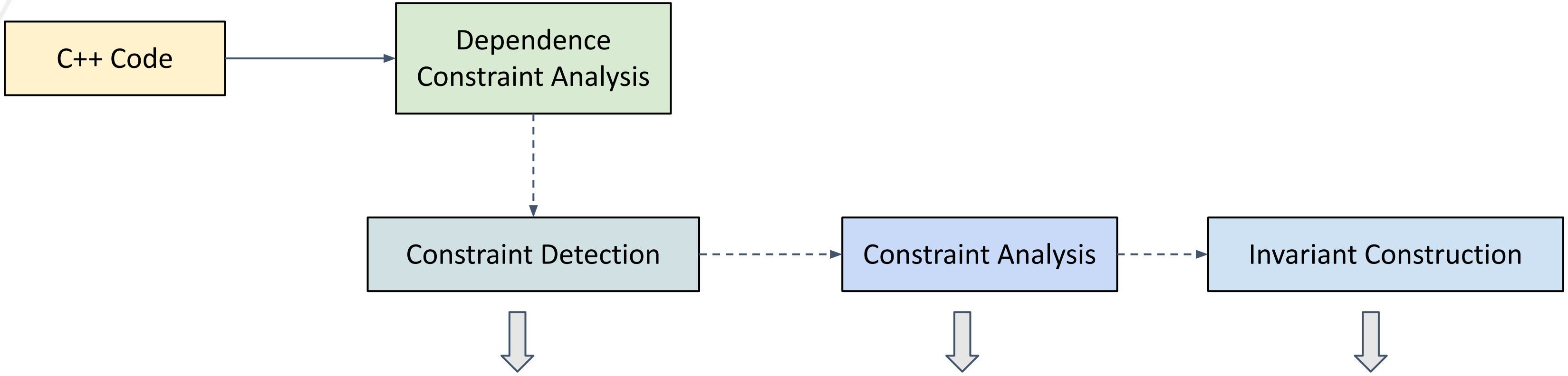
```
procedure main () {
var $i, $x, $MAXSIZE, $check, $r, $temp, $sub: int;
$bb0:
    $i := 0;
    $x := 0;
    goto $bb1;
$bb1:
    $temp := $slt.i32($i, $MAXSIZE);
    assume {:branchcond $temp} true;
    goto $bb2, $bb4;
$bb2:
    $x := $i;
    goto $bb3;
$bb3:
    $i := $add.i32($i, 1);
    goto $bb1;
$bb4:
    $sub := $sub.i32($MAXSIZE, 1);
    $check := $eq.i32($x, $sub);
    assume {:branchcond $check} true;
    goto $bb7, $bb6;
$bb6:
    call __VERIFIER_assert(0);
    goto $bb7
$bb7:
    $r := 1;
    return;
}
```

Boogie IVL

Same Assertion Check

Counter-example that bb4 will never go to bb6
so the assertion is true

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

Dependence Constraint Analysis

Constraint Detection → Constraint Analysis → Invariant Construction

```
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;
}
```

Original C/C++ Code

Derived

Absolute

```
-j+j'-k1 = 0
-i+i'+k2 = 0
-k2-1 >= 0
k1-1 >= 0
j+k1-1 >= 0
-j+j'+k2 = 0
-i+i'-k1 = 0
-t+t'-1 >= 0
-t'+999 >= 0
i+k1-1 >= 0
j-k2-1 >= 0
...
```

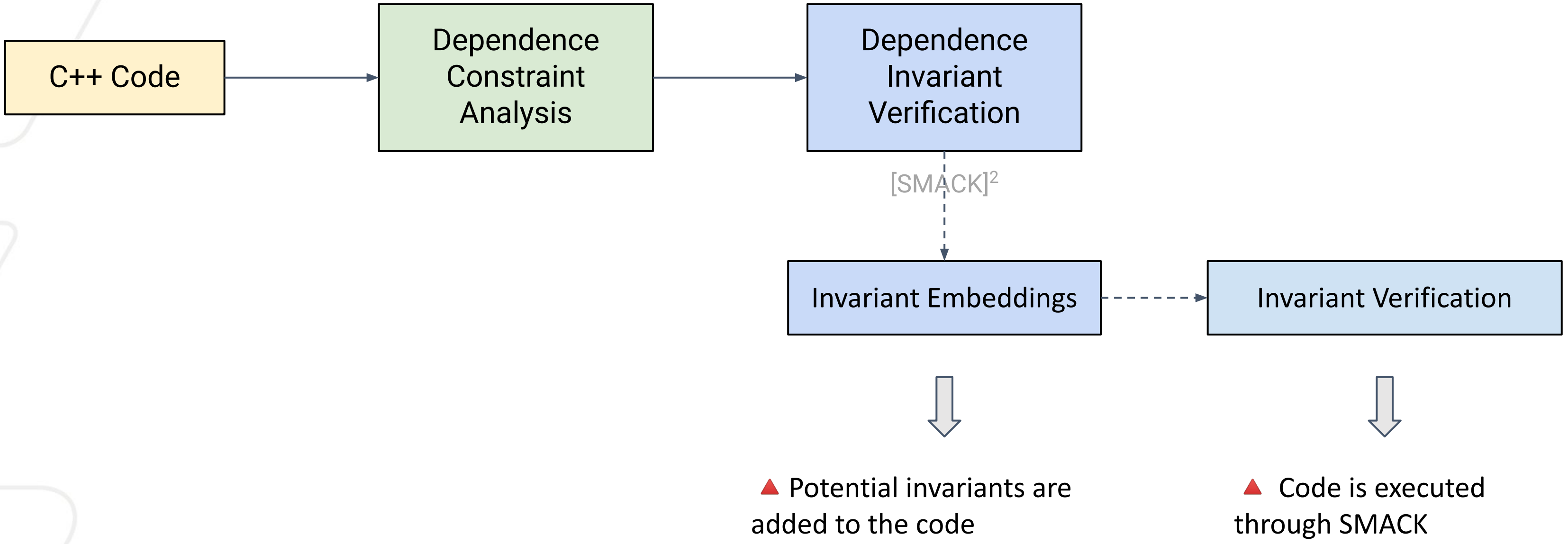
39 Data-Dependencies with
30 Optimization Constraints (24 derived and 6 absolute)

```
k2 > 1
k1 < 1
k1 <= 1
k2 > N
..
```

All derived constraints are converted to absolute constraints

Constraints are converted to potential invariants

VICO: Demand-Driven Verification for Improving Compiler Optimization



Georgia Tech

Dependence Invariant Verification

Invariant Embeddings → Invariant Verification

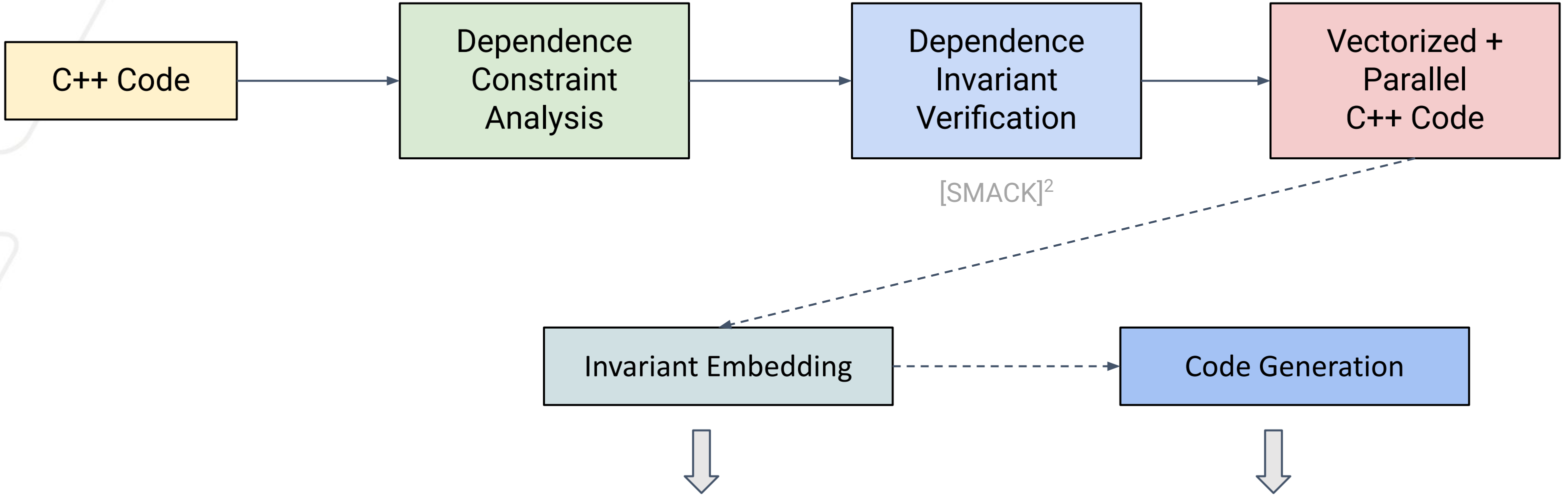
```
void liebmann2D (/*arguments*/) {

int k1 = getK(N), k2 = getK(N);
for (t = 0; t <= M; t++)
 assert(k2 > N);
 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;
}
```

```
SMACK found no errors
k2 > N is an invariant
```

Original C/C++ Code with a potential Invariant

Invariant Verification with 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

Vectorized + Parallelized C++ Code

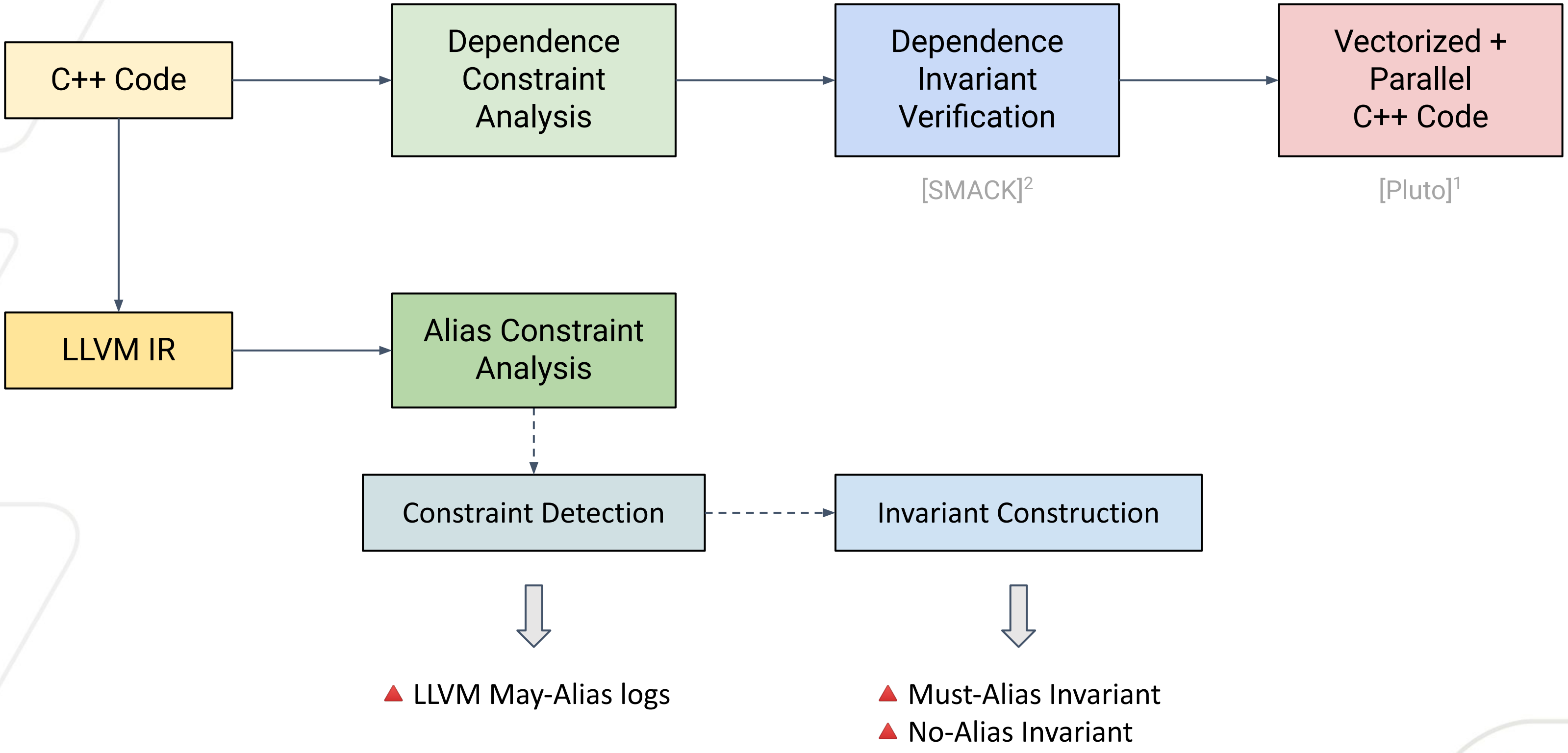

```
void liebmann2D (/*arguments*/) {

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
```

```
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;
}
```

Embedded C/C++ Code

Parallelized C/C++ Code

VICO: Demand-Driven Verification for Improving Compiler Optimization



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

Alias Constraint Analysis

Constraint Detection → Constraint Analysis → Invariant Construction

```c
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){
   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

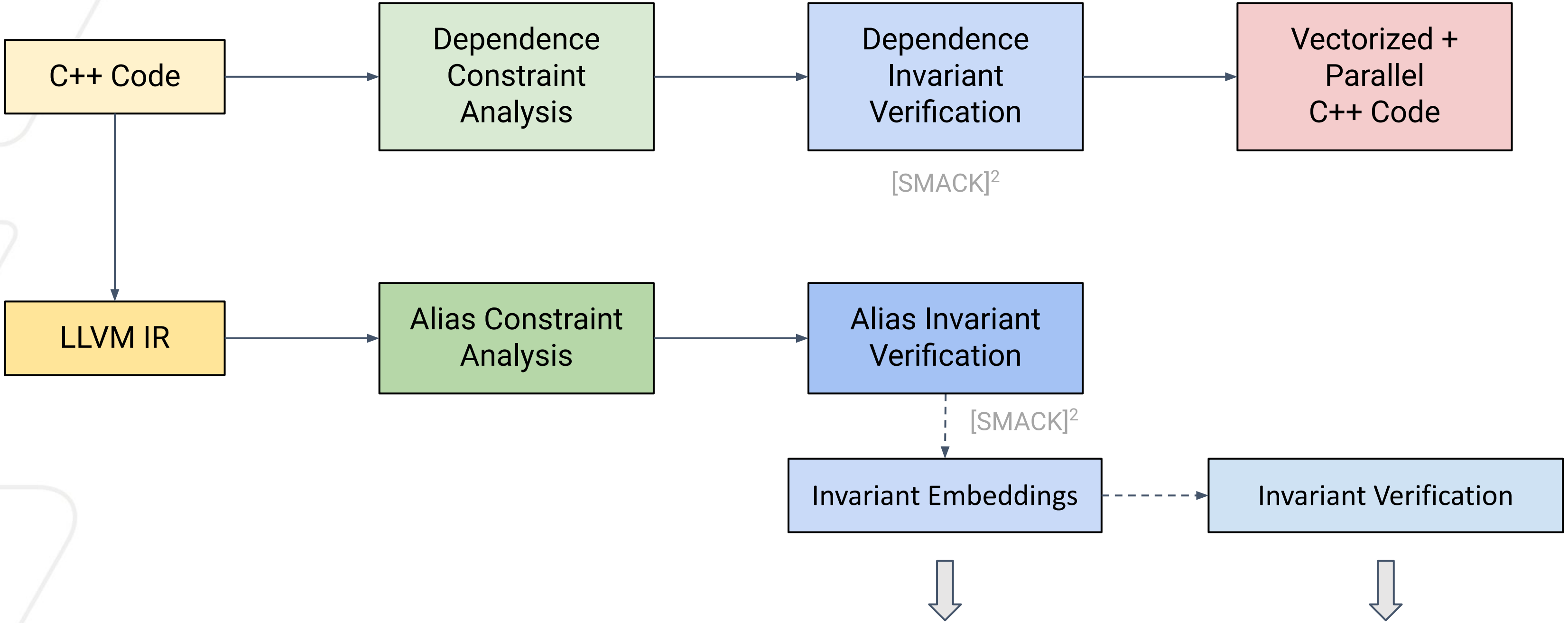
```
p ≟ &i
p ≟ &j
p ≟ &k
p ≟ &l
```

4 optimization constraints

```
p = &i
p = &j
p = &k
p = &l
p ≠ &i
p ≠ &j
p ≠ &k
p ≠ &l
```

Constraints are converted to potential invariants

VICO: Demand-Driven Verification for Improving Compiler Optimization



- Must Alias Invariants Embedded
- No Alias Invariants Embedded

- Code is executed through SMACK

2. https://github.com/smackers/smack

Alias Invariant Analysis

Invariant Embeddings → Invariant Verification

```
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);
  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 Must Alias Invariant

```
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);
  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 */}
```

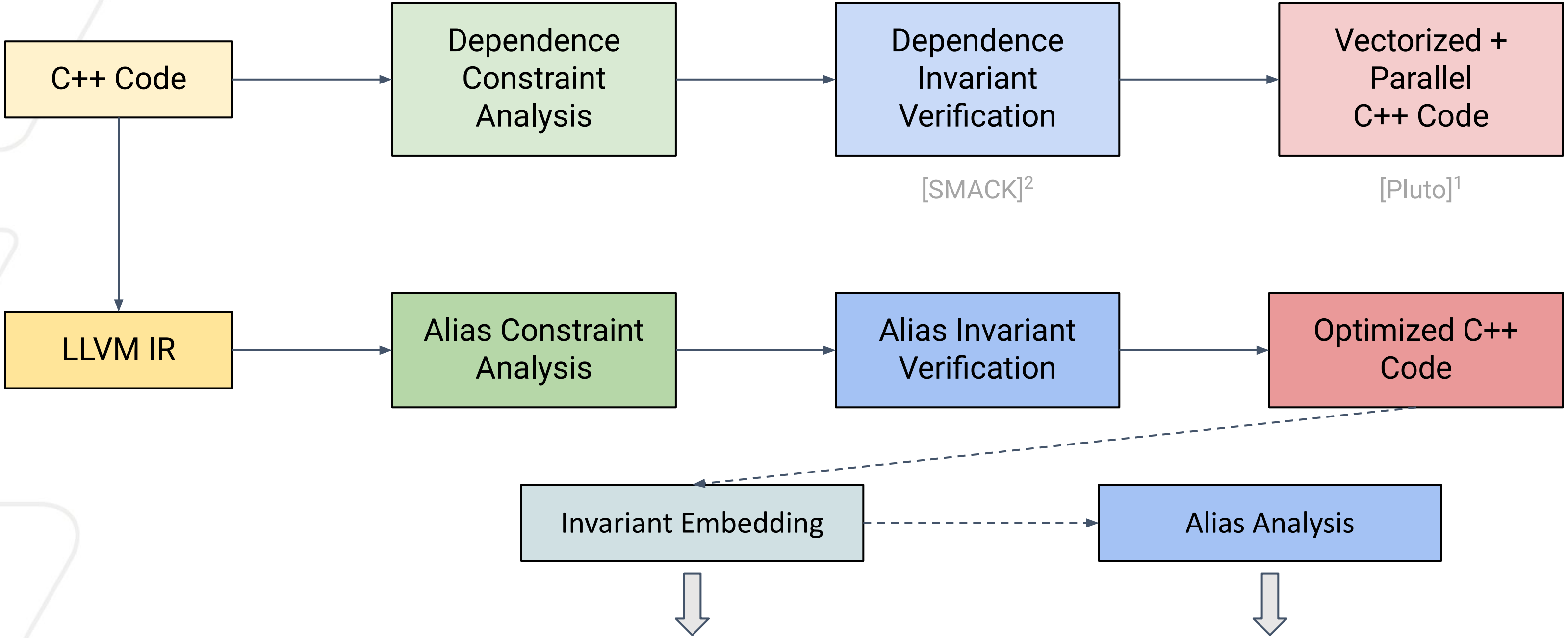
Original C/C++ Code with No Alias Invariant

```
p = &l is verified
p != &l is not verified
```

Invariant Verification with 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

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

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

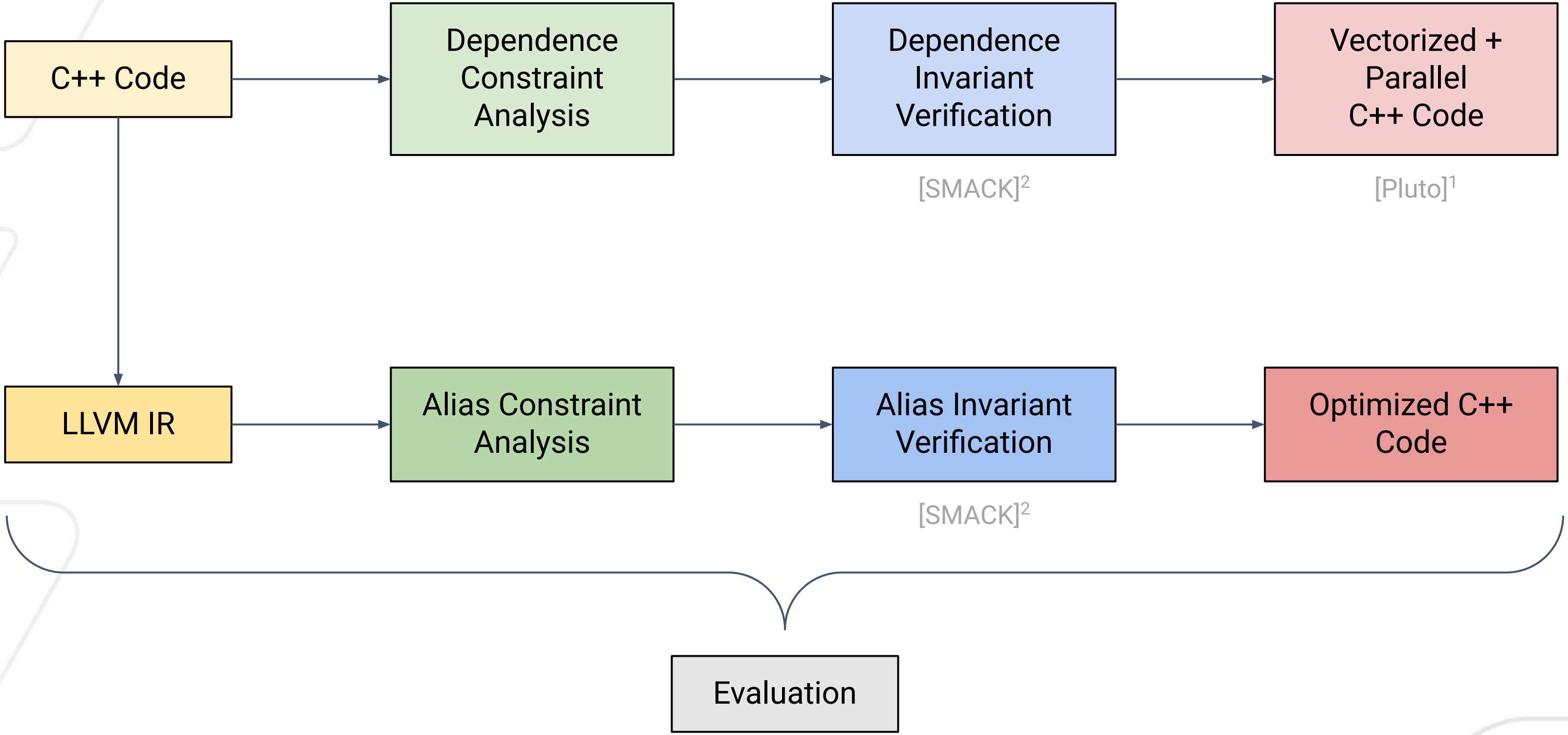
```
%3 ≠ null (p = &l)
%3 ≠ %6 (p != &k)
%3 ≠ %5 (p != &j)
%3 ≠ %4 (p != &i)
```

Our Alias Analysis saving the invariants

↓

LLVM's Alias Analysis

Evaluation



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

Summary of Results

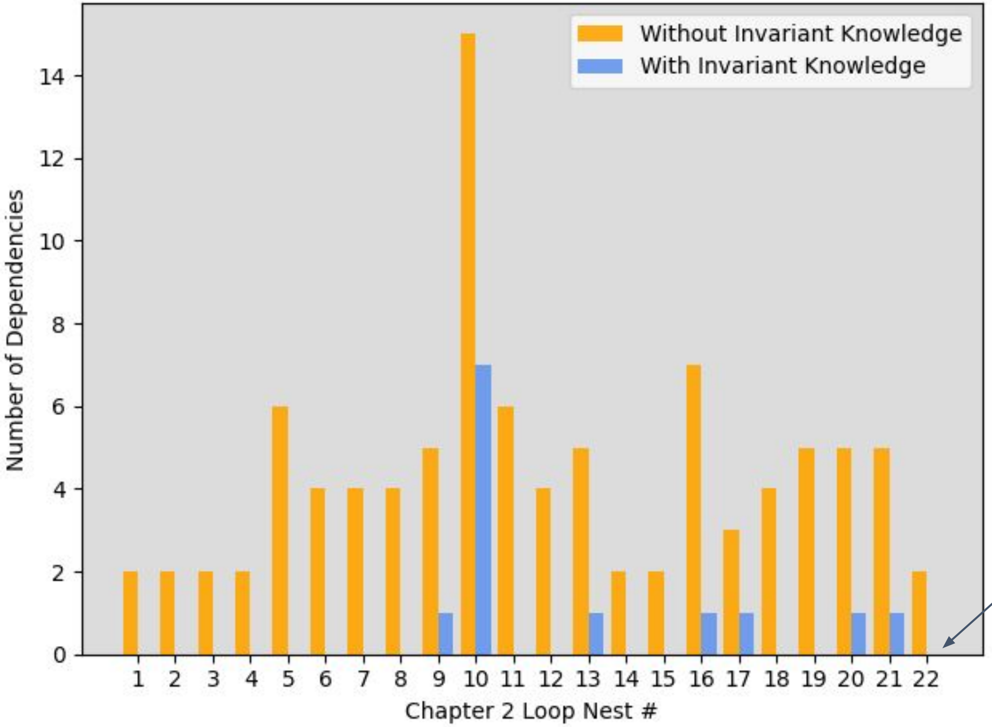
- Improving precision of dependence analysis by 45% in real-world cases
  - Better optimizations in over 75 loops
  - Average speed-up of 14.7x on Apple M1 Pro
  - Average speed-up of 6.07x on Intel Xeon E5-2660
  - Took a total time of more than 5 hours
- Improving precision of alias analysis
  - Average 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 cases

Georgia Tech

Benchmarks

- Source-to-Source Parallelization Improvement
  - Kernel Programs (From Kennedy et al. book) combined with Invariant implementations from Si et al.
  - Mathematical Applications from Polybench adapted with generalized boundaries
- Backend Compiler Optimizations Improvement
  - Micro-benchmarks useful to force may-alias cases
  - Real-world applications from SPEC 2017 and CoreUtils

Kernel Programs



0 Dependencies in 15 loop nests

Mathematical Applications

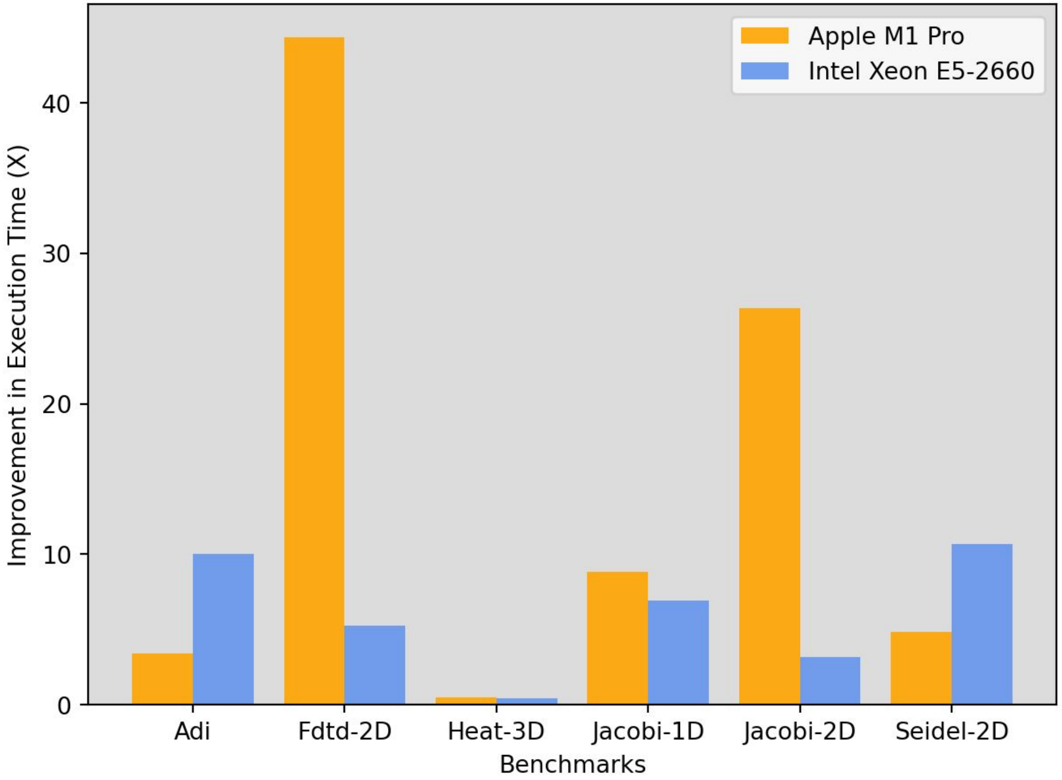
Applications	Potential Invariants		Data-Dependencies	
	Absolute Invariants	Derived Invariants	Without Invariant Knowledge	With Invariant Knowledge
Alternating Direction Implicit method with generalized shift parameters	15	56	214	118
Multi-dimensional Finite Difference Time Domain	7	48	38	28
Heat Equation in three dimensions with artificial boundary conditions in unbounded domain	0	54	106	42
Jacobi Iterative Method in one dimension with generalized boundary conditions	0	18	14	14
Jacobi Iterative Method in two dimensions with generalized boundary conditions	0	36	22	22
Liebmann's Method in two dimensions with generalized boundary conditions	6	24	39	19

Georgia Tech

Mathematical Applications

Applications	Loop Optimizations	
	Without Invariant Knowledge	With Invariant Knowledge
Alternating Direction Implicit method with generalized shift parameters	Serial Loop, Serial Loop, Serial Loop	Serial Loop, Parallel Loop, Serial Loop + Loop Splitting
Multi-dimensional Finite Difference Time Domain	Serial Loop, Serial Loop, Parallel Loop, Parallel Loop + Loop Splitting	Serial Loop, Parallel Loop Parallel Loop, Parallel Loop + Loop Splitting
Heat Equation in three dimensions with artificial boundary conditions in unbounded domain	Serial Loop, Serial Loop, Serial Loop	Parallel Loop, Vectorized Loop, Vectorized Loop + Loop Splitting
Jacobi Iterative Method in one dimension with generalized boundary conditions	Serial Loop, Serial Loop, Serial Loop	Parallel Loop, Parallel Loop, Parallel Loop + Loop Splitting
Jacobi Iterative Method in two dimensions with generalized boundary conditions	Serial Loop, Serial Loop, Parallel Loop + Loop Splitting	Serial Loop, Parallel Loop, Parallel Loop + Loop Splitting
Liebmann's Method in two dimensions with generalized boundary conditions	Serial Loop, Serial Loop	Serial Loop, Parallel Loop

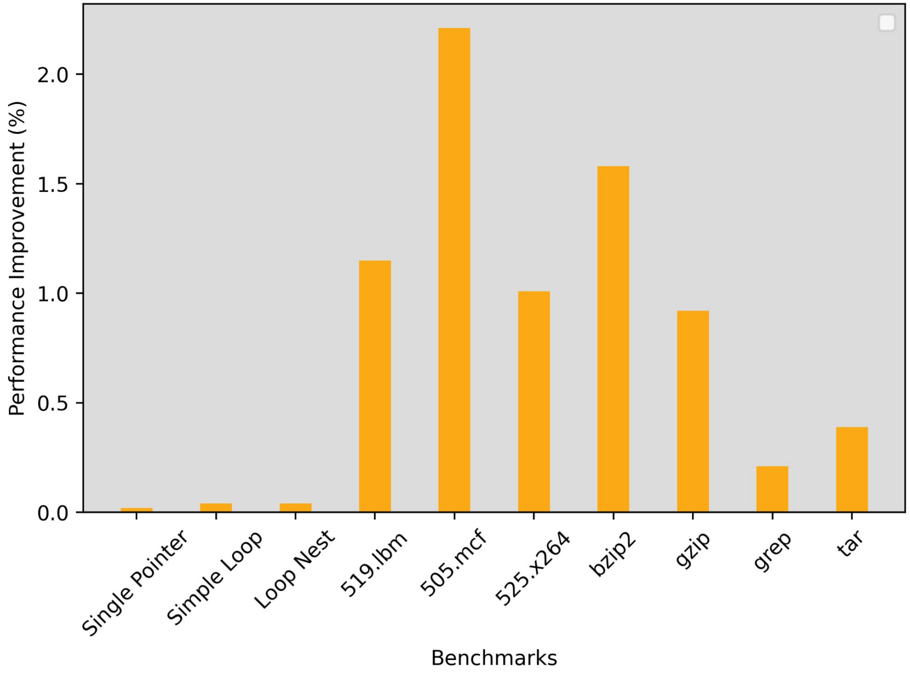
Performance Improvement

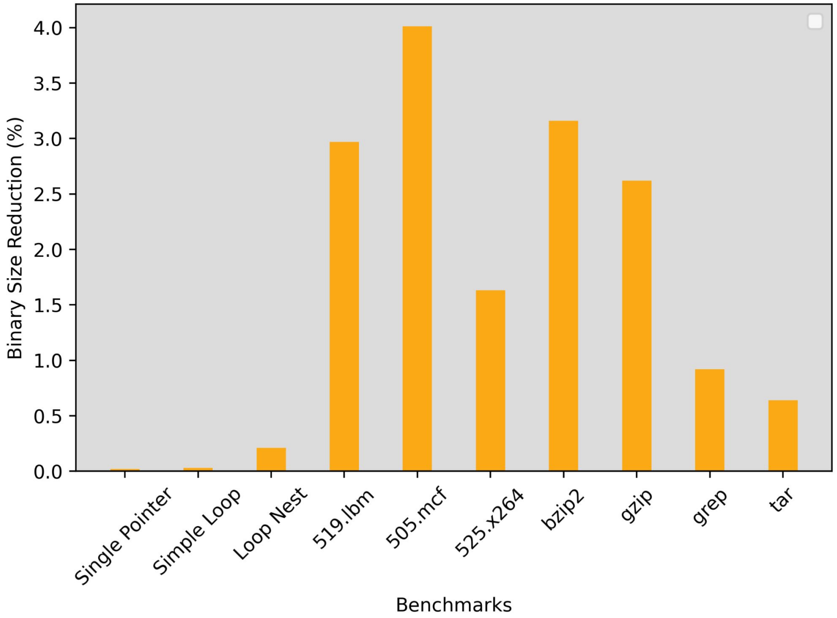


Backend Results

	Number of Constraints	Number of Verified Must-Alias	Number of Verified No-Alias	Changes in Value Numbering	New PRE Removed Redundancies
Single Pointer	1	1	0	1	0
Simple Loop	4	1	3	2	2
Loop Nest	5	4	1	3	1
LBM	9	2	2	2	1
MCF	22	6	2	6	4
X264	13	4	0	4	0
Bzip2	14	4	1	4	2
Gzip	9	3	3	3	3
Grep	7	0	3	0	0
Tar	9	1	1	1	0

Backend Applications





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