VICO: Demand-Driven Verification for Improving Compiler Optimizations

Sharjeel Khan    Bodhisatwa Chatterjee    Santosh Pande
Motivation: Traditional Compiler Analysis suffer from Imprecision

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

- Data-Dependence Analysis
- Alias Analysis

Conservative (Safe) Approximations

- Inability to check infeasible program paths
- Inability to symbolically propagate and evaluate expressions
- Inability to verify statically unknown properties \textit{inter-procedurally}
Example: Liebmann’s Method with generalized boundary conditions

```c
int getK (int par) {
    if (par % 2)
        return 2*(par + 1);
    else
        return 2*par;
}

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 + k2][j - k1] + A[i + k2][j]
                          + A[i + k2][j + k2])/c;
}
```

Possible implementation of boundary offset values

k1 and k2 initialized by external function calls

Interprocedural whole program flow analysis unable to prove the \((k_1, k_2 > N)\) invariant

Proving \((k_1, k_2 > N)\) breaks most dependencies

\(0 \leq k_1, k_2 \leq N\) (Dependence Equation)

Compiler assumes all possible dependencies
Demand-driven verification based solution

Proving Optimization Constraints

- Proves only those properties that are related to optimization instance at hand
- Has the ability to pick properties that can break maximum constraints

Demand-Driven Verification

Use of Software Verification in Compilers

- 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

Verification for Compiler Optimizations

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

C++ Code → Dependence Constraint Analysis → Constraint Detection → Constraint Analysis → Invariant Construction

- PLuTo dependence logs
- Detecting absolute & derived constraints
- Constraint transformation
- Constraint selection
- Operator flipping

1. https://github.com/bondhugula/pluto
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
VICO: Demand-Driven Verification for Improving Compiler Optimization

C++ Code

Dependence Constraint Analysis

Dependence Invariant Verification

Vectorized + Parallel C++ Code

[SMACK]$^2$

[Pluto]$^1$

LLVM IR

Alias Constraint Analysis

Constraint Detection

Invariant Construction

🔺 LLVM May-Alias logs

🔺 Must-Alias Invariant

🔺 No-Alias Invariant

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
VICO: Demand-Driven Verification for Improving Compiler Optimization

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

C++ Code → Dependence Constraint Analysis → Dependence Invariant Verification → Vectorized + Parallel C++ Code

LLVM IR → Alias Constraint Analysis → Alias Invariant Verification → Optimized C++ Code

Invariant Embedding

- 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

[SMACK]²

[Pluto]¹
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
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
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 + k2][j + k1] + A[i][j + k2] + k2
                     + A[i][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(ceil(t+1, 2), t-M+1);
    ubp = min(floor(t+1, N, 0), t);
    #pragma omp parallel for private(lbv,ubv)
    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-1][-t+j-1] + A[-t+2*i][(-t+j)+1]
                          + A[-t+2*i][-t+j+1]
                          + A[-t+2*i][(-t+j)+1] + A[-t+2*i][(-t+j)+1]
                          + A[-t+2*i][(-t+j)+1] + A[-t+2*i][(-t+j)+1] / c;
      }
    }
  }
}
```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) {
        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 */
}
```

Our Alias Analysis saving the invariants

LLVM’s Alias Analysis

define dso_local i32 @main(i32 %0, i8** %1) #2 !dbg !356 {
    %50 = icmp ne i32 * %3, %6, !dbg !430, !verifier.code !344
    br if %51, label %53, label %52, !dbg !433, !verifier.code !344
    %52 = icmp ne i32 * %3, %6, !dbg !430, !verifier.code !344
    call void @__VERIFIER_assert(i32 0), !dbg !430, !verifier.code !428
    br label %53, !dbg !430, !verifier.code !344
    "...
    %62 = icmp ne i32 * %3, %4, !dbg !440, !verifier.code !344
    br if %63, label %65, label %64, !dbg !443, !verifier.code !344
    %64 = icmp ne i32 * %3, %6, !dbg !440, !verifier.code !344
    call void @__VERIFIER_assert(i32 0), !dbg !440, !verifier.code !428
    br label %65, !dbg !440, !verifier.code !344
}