VICO: Demand-Driven Verification for Improving Compiler Optimizations

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

- Dependence Analysis
- Alias Analysis

**Code Optimization Techniques**

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

**Data-Dependence Analysis**

**Alias Analysis**

- 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
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

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

```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 \((k1,k2 > N)\) 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)

```c
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
Key Insight: “Verification can boost Compiler Optimizations”

Use of Software Verification in Compilers

- **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

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
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
```c
int main() {
    int x = 0, i = 0;
    for (i = 0; i < M; ++i) {
        x = i;
    }
    assert (x == M - 1);
    return 0;
}
```
VICO: Demand-Driven Verification for Improving Compiler Optimization

1. https://github.com/bondhugula/pluto
Dependence Constraint Analysis

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++)
}

Original C/C++ Code

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

Constraint Detection

Constraint Analysis

Invariant Construction

- \( j + j' - k1 = 0 \)
- \( i + i' + k2 = 0 \)
- \( k2 - 1 \geq 0 \)
- \( k1 - 1 \geq 0 \)
- \( j + k1 - 1 \geq 0 \)
- \( -j + j' + k2 = 0 \)
- \( -i + i' - k1 = 0 \)
- \( -t + t' - 1 \geq 0 \)
- \( i + k1 - 1 \geq 0 \)
- \( j - k2 - 1 \geq 0 \)

All derived constraints are converted to absolute constraints

Constraints are converted to potential invariants
Dependence 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 + k2][j - k1] + A[i + k2][j]
                       + A[i + k2][j + k2]) / c;
}

Original C/C++ Code
with a potential
Invariant

SMACK found no errors
k2 > N is an invariant

Invariant Verification

SMACK found no errors
k2 > N is an invariant

Invariant Verification
with Smack
VICO: Demand-Driven Verification for Improving Compiler Optimization

- **C++ Code**
  - Dependence Constraint Analysis
  - Dependence Invariant Verification
  - Vectorized + Parallel C++ Code

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

**Code Generation**
- PLuTo\(^1\) generates optimized code from the embedded code

[SMAACK]\(^2\)

1. https://github.com/bondhugula/pluto
2. https://github.com/smackers/smack
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 + k2][j]
                        + A[i + k2][j + k2]) / c;
    }
    #pragma endscop
}

void liebmann2D /*arguments*/
{
    int k1 = getK(N), k2 = getK(N);
    #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)][(-t+j)-1]
                + A[(-t+2*i)][(-t+j)] + A[(-t+2*i)][(-t+j)+1]
                + A[(-t+2*i)+1][(-t+j)-1] + A[(-t+2*i)+1][(-t+j)]
                + A[(-t+2*i)+1][(-t+j)+1]) / c;
}
VICO: Demand-Driven Verification for Improving Compiler Optimization

1. https://github.com/bondhugula/pluto
2. https://github.com/smackers/smack
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 */
}
VICO: Demand-Driven Verification for Improving Compiler Optimization

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

- LLVM IR
  - Alias Constraint Analysis
  - Alias Invariant Verification
    - Invariant Embeddings
      - Must Alias Invariants Embedded
      - No Alias Invariants Embedded
  - Invariant Verification
    - Code is executed through SMACK

2. https://github.com/smackers/smack
Alias Invariant Analysis

```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);
        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

```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);
        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

```
Invariant Embeddings
```

```
Invariant Verification with Smack
```

p = &l is verified
p != &l is not verified
VICO: Demand-Driven Verification for Improving Compiler Optimization

- **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

1. https://github.com/bondhugula/pluto
2. https://github.com/smackers/smack
Optimized C++ Code

**Invariant Embedding**

```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 */
}
```

**Alias Analysis**

```c
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, %5, !dbg !435, !verifier.code !344
    call void @__VERIFIER_assert(i32 0), !dbg !430, !verifier.code !428
    label %53, !dbg !430, !verifier.code !344
    %55 = icmp ne i32 *%3, %4, !dbg !435, !verifier.code !344
    call void @__VERIFIER_assert(i32 0), !dbg !435, !verifier.code !428
    label %56, !dbg !435, !verifier.code !344
    %61 = icmp ne i32 *%3, %4, !dbg !440, !verifier.code !344
    call void @__VERIFIER_assert(i32 0), !dbg !440, !verifier.code !428
    label %62, !dbg !440, !verifier.code !344
    %65 = icmp ne i32 *%3, %5, !dbg !443, !verifier.code !344
    call void @__VERIFIER_assert(i32 0), !dbg !443, !verifier.code !428
    label %66, !dbg !440, !verifier.code !344
}
```

**LLVM IR representation**

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

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

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

[SMACK]$^2$

[Pluto]$^1$

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
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

Graph showing number of dependencies in 15 loop nests.

- Without Invariant Knowledge
- With Invariant Knowledge

0 Dependencies in 15 loop nests
# Mathematical Applications

<table>
<thead>
<tr>
<th>Applications</th>
<th>Potential Invariants</th>
<th>Data-Dependencies</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Absolute Invariants</td>
<td>Derived Invariants</td>
</tr>
<tr>
<td>Alternating Direction Implicit method with generalized shift parameters</td>
<td>15</td>
<td>56</td>
</tr>
<tr>
<td>Multi-dimensional Finite Difference Time Domain</td>
<td>7</td>
<td>48</td>
</tr>
<tr>
<td>Heat Equation in three dimensions with artificial boundary conditions in unbounded domain</td>
<td>0</td>
<td>54</td>
</tr>
<tr>
<td>Jacobi Iterative Method in one dimension with generalized boundary conditions</td>
<td>0</td>
<td>18</td>
</tr>
<tr>
<td>Jacobi Iterative Method in two dimensions with generalized boundary conditions</td>
<td>0</td>
<td>36</td>
</tr>
<tr>
<td>Liebmann’s Method in two dimensions with generalized boundary conditions</td>
<td>6</td>
<td>24</td>
</tr>
<tr>
<td>Applications</td>
<td>Loop Optimizations Without Invariant Knowledge</td>
<td>Loop Optimizations With Invariant Knowledge</td>
</tr>
<tr>
<td>------------------------------------------------------------------------------</td>
<td>-----------------------------------------------</td>
<td>-----------------------------------------------------</td>
</tr>
<tr>
<td><strong>Alternating Direction Implicit method with generalized shift parameters</strong></td>
<td>Serial Loop, Serial Loop, Serial Loop</td>
<td>Serial Loop, Parallel Loop, Serial Loop + Loop Splitting</td>
</tr>
<tr>
<td><strong>Multi-dimensional Finite Difference Time Domain</strong></td>
<td>Serial Loop, Serial Loop, Parallel Loop,</td>
<td>Serial Loop, Parallel Loop Parallel Loop + Loop Splitting</td>
</tr>
<tr>
<td></td>
<td>Parallel Loop + Loop Splitting</td>
<td></td>
</tr>
<tr>
<td><strong>Heat Equation in three dimensions with artificial boundary conditions in</strong></td>
<td>Serial Loop, Serial Loop, Serial Loop</td>
<td>Parallel Loop, Vectorized Loop, Vectorized Loop + Loop Splitting</td>
</tr>
<tr>
<td>unbounded domain</td>
<td></td>
<td></td>
</tr>
<tr>
<td>**Jacobi Iterative Method in one dimension with generalized</td>
<td>Serial Loop, Serial Loop, Serial Loop</td>
<td>Parallel Loop, Parallel Loop, Parallel Loop Splitting</td>
</tr>
<tr>
<td>boundary conditions**</td>
<td></td>
<td></td>
</tr>
<tr>
<td>**Jacobi Iterative Method in two dimensions with generalized</td>
<td>Serial Loop, Serial Loop, Parallel Loop +</td>
<td>Serial Loop, Parallel Loop, Parallel Loop Splitting</td>
</tr>
<tr>
<td>boundary conditions**</td>
<td>Loop Splitting</td>
<td></td>
</tr>
<tr>
<td><strong>Liebmann’s Method in two dimensions with generalized boundary conditions</strong></td>
<td>Serial Loop, Serial Loop</td>
<td>Serial Loop, Parallel Loop</td>
</tr>
</tbody>
</table>
Performance Improvement

![Performance Improvement Chart](image-url)
### Backend Results

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