Optimistic Loop Optimization

CGO 2017 – February 8th – Austin, TX

Johannes Doerfert and Sebastian Hack
Compiler Design Lab
Saarland University
http://compilers.cs.uni-saarland.de

Tobias Grosser
Department of Computer Science
ETH Zürich
https://spcl.inf.ethz.ch

Motivating Example

A POTENTIALLY PARALLEL LOOP

```
for (i = 0; i < N; i++)
  A[i] = A[N + i];
```

A POTENTIALLY PARALLEL LOOP

```
for (i = 0; i < N; i++)
  A[i] = A[N + i];
```

Read Set ($\mathbf{R}$)

$\{ A[N + i] \mid 0 \leq i < N \}$

Write Set ($\mathbf{W}$)

$\{ A[i] \mid 0 \leq i < N \}$

A POTENTIALLY PARALLEL LOOP

```
for (i = 0; i < N; i++)
  A[i] = A[N + i];
```

Read Set (**R**)

$\{ \texttt{A[N + i]} \mid 0 \leq \texttt{i} < \texttt{N} \}$

Write Set (**W**)

$\{ \texttt{A[i]} \mid 0 \leq \texttt{i} < \texttt{N} \}$

$$\texttt{R} \cap \texttt{W} = \{ \}$$

A POTENTIALLY PARALLEL LOOP

```
for (i = 0; i < N; i++)
  A[i] = A[N + i];
```

Read Set ($\mathtt{R}$)

$\{\ \mathtt{A[N + i]} \mid 0 \leq \mathtt{i} < \mathtt{N}\ \}$

Write Set ($\mathtt{W}$)

$\{\ \mathtt{A[i]} \mid 0 \leq \mathtt{i} < \mathtt{N}\ \}$

$\mathtt{R} \cap \mathtt{W} = \{\ \}$ *Parallel*

A POTENTIALLY PARALLEL LOOP

```
unsigned char i, N;

for (i = 0; i < N; i++)
  A[i] = A[N + i];
```

Read Set (**R**)

$\{ A[N + i] \mid 0 \leq i < N \}$

Write Set (**W**)

$\{ A[i] \mid 0 \leq i < N \}$

A POTENTIALLY PARALLEL LOOP

```
unsigned char i, N;

for (i = 0; i < N; i++)
  A[i] = A[N + i];
```

Read Set (**R**)

~~$\{ \texttt{A[N + i]} \mid 0 \leq \texttt{i} < \texttt{N} \}$~~

Write Set (**W**)

$\{ \texttt{A[i]} \mid 0 \leq \texttt{i} < \texttt{N} \}$

A POTENTIALLY PARALLEL LOOP

```
unsigned char i, N;

for (i = 0; i < N; i++)
  A[i] = A[N + i];
```

Read Set (**R**)

~~$\{\ A[N + i] \mid 0 \leq i < N\ \}$~~

$\{\ A[(N + i) \bmod 256] \mid \dots\ \}$

Write Set (**W**)

$\{\ A[i] \mid 0 \leq i < N\ \}$

A Potentially Parallel Loop

```
unsigned char i, N;

for (i = 0; i < N; i++)
  A[i] = A[N + i];
```

Read Set (R)

~~{ A[N + i] | 0 ≤ i < N }~~

{ A[(N + i) mod 256] | … }

Write Set (W)

{ A[i] | 0 ≤ i < N }

R ∩ W = { }, iff N <= 128

A Potentially Parallel Loop

```
unsigned char i, N;

for (i = 0; i < N; i++)
  A[i] = A[N + i];
```

Read Set (**R**)

~~{ A[N + i] | 0 ≤ i < N }~~

{ A[(N + i) mod 256] | … }

Write Set (**W**)

{ A[i] | 0 ≤ i < N }

R ∩ W = { }, **iff** N <= 128

Potentially Sequential

Problem Statement

PROBLEM STATEMENT

Required:
Program abstractions that capture *all possible semantics*

PROBLEM STATEMENT

Required:
Program abstractions that capture *all possible semantics*

Reality:
Corner cases are often *missed* or *assumed* not to happen

PROBLEM STATEMENT

Required:
Program abstractions that capture *all possible semantics*

Reality:
Corner cases are often *missed* or *assumed* not to happen

Consequence:
Poor applicability and *miscompilations* for certain inputs

PROBLEM STATEMENT

Required:
Program abstractions that capture *all possible semantics*

Reality:
Corner cases are often *missed* or *assumed* not to happen

Consequence:
Poor applicability and *miscompilations* for certain inputs

Solution:
Take *optimistic assumptions statically* that are *verified dynamically*

Solution

OPTIMISTIC LOOP OPTIMIZATION

OPTIMISTIC LOOP OPTIMIZATION

```
/* loop nest */
```

OPTIMISTIC LOOP OPTIMIZATION

1. Take *Optimistic Assumptions* to model the loop nest

```
/* loop nest */
```

OPTIMISTIC LOOP OPTIMIZATION

1. Take *Optimistic Assumptions* to model the loop nest
2. Optimize the loop nest

```
/* optimized loop nest */

/* loop nest */
```

OPTIMISTIC LOOP OPTIMIZATION

1. Take *Optimistic Assumptions* to model the loop nest
2. Optimize the loop nest
3. Version the code

```
if (                          )
  /* optimized loop nest */
else
  /* loop nest */
```

OPTIMISTIC LOOP OPTIMIZATION

1. Take *Optimistic Assumptions* to model the loop nest
2. Optimize the loop nest
3. Version the code
4. Create a *simple* runtime check

```
if (/* simple runtime check */)
  /* optimized loop nest */
else
  /* loop nest */
```

SEMANTIC DIFFERENCES

C	LLVM-IR	Polyhedral Model

SEMANTIC DIFFERENCES

C	LLVM-IR	Polyhedral Model
Variant Loads in Control Conditions		
✓	✓	✗

SEMANTIC DIFFERENCES

C	LLVM-IR	Polyhedral Model
Variant Loads in Control Conditions		
✓	✓	✗
Aliasing Arrays		
✓	✓	✗

SEMANTIC DIFFERENCES

C	LLVM-IR	Polyhedral Model
Variant Loads in Control Conditions		
✓	✓	✗
Aliasing Arrays		
✓	✓	✗
Integer Wrapping		
✓	✓	✗

SEMANTIC DIFFERENCES

C	LLVM-IR	Polyhedral Model
Variant Loads in Control Conditions		
✓	✓	✗
Aliasing Arrays		
✓	✓	✗
Integer Wrapping		
✓	✓	✗
Out-of-Bound Accesses		
✓	✓	✗

SEMANTIC DIFFERENCES

	C	LLVM-IR	Polyhedral Model
Variant Loads in Control Conditions	✓	✓	✗
Aliasing Arrays	✓	✓	✗
Integer Wrapping	✓	✓	✗
Out-of-Bound Accesses	✓	✓	✗
Potentially Unbounded Loops	✓	✓	✗

Real World Example

REAL WORLD EXAMPLE

NAS Parallel Benchmark Suite – BT – compute_rhs

- 66 loops, nested up to depth 4
- 38 array writes, 294 array reads
- 45 reads in loop bounds

REAL WORLD EXAMPLE

```
double rhs[JMAX][IMAX][5];



for (j = 0; j < grid[0] + 1; j++)

  for (i = 0; i < grid[1] + 1; i++)
    for (m = 0; m < 5; m++)




      rhs[j][i][m] = /* ... */;
```

ASSUMPTION GENERATION

```
double rhs[JMAX][IMAX][5];



for (j = 0; j < grid[0] + 1; j++)

  for (i = 0; i < grid[1] + 1; i++)
    for (m = 0; m < 5; m++)




      rhs[j][i][m] = /* ... */;
```

(a) Loads in control and access functions are invariant

ASSUMPTION GENERATION

```
double rhs[JMAX][IMAX][5];



for (j = 0; j < grid[0] + 1; j++)

  for (i = 0; i < grid[1] + 1; i++)
    for (m = 0; m < 5; m++)




      rhs[j][i][m] = /* ... */;
```

(a) Loads in control and access functions are invariant

(b) No aliasing/overlapping arrays

ASSUMPTION GENERATION

```
double rhs[JMAX][IMAX][5];



for (j = 0; j < grid[0] + 1; j++)

  for (i = 0; i < grid[1] + 1; i++)
    for (m = 0; m < 5; m++)

      assume &rhs[j][i][m] >= &grid[2] ||
             &rhs[j][i][m + 1] <= &grid[0];
      rhs[j][i][m] = /* ... */;
```

(a) Loads in control and access functions are invariant

(b) No aliasing/overlapping arrays

ASSUMPTION GENERATION

```
double rhs[JMAX][IMAX][5];



for (j = 0; j < grid[0] + 1; j++)

  for (i = 0; i < grid[1] + 1; i++)
    for (m = 0; m < 5; m++)

      assume &rhs[j][i][m] >= &grid[2] ||
             &rhs[j][i][m + 1] <= &grid[0];
      rhs[j][i][m] = /* ... */;
```

(c) Expressions do not wrap

Assumption Generation

```
double rhs[JMAX][IMAX][5];

assume grid[0] != MAX_VALUE;
for (j = 0; j < grid[0] + 1; j++)
  assume grid[1] != MAX_VALUE;
  for (i = 0; i < grid[1] + 1; i++)
    for (m = 0; m < 5; m++)

      assume &rhs[j][i][m] >= &grid[2] ||
             &rhs[j][i][m + 1] <= &grid[0];
      rhs[j][i][m] = /* ... */;
```

(c) Expressions do not wrap

NO WRAPPING ASSUMPTIONS

Given an expression e with m bits:

(c) Expressions do not wrap

NO WRAPPING ASSUMPTIONS

Given an expression e with m bits:

$$[\![\mathsf{e}]\!]_{\mathbb{Z}}$$

(c) Expressions do not wrap

No Wrapping Assumptions

Given an expression e with m bits:

$$[\![\mathrm{e}]\!]_{\mathbb{Z}} \qquad [\![\mathrm{e}]\!]_{\mathbb{Z}_{2^m}/\mathbb{Z}}$$

(c) Expressions do not wrap

NO WRAPPING ASSUMPTIONS

Given an expression e with m bits:

$$[\![\mathsf{e}]\!]_{\mathbb{Z}} \neq [\![\mathsf{e}]\!]_{\mathbb{Z}_{2^m}/\mathbb{Z}}$$

(c) Expressions do not wrap

NO WRAPPING ASSUMPTIONS

Given an expression e with m bits:

$$\mathcal{I}_W(\mathsf{e}) = \{(\underline{i}) \mid [\![\mathsf{e}]\!]_{\mathbb{Z}} \neq [\![\mathsf{e}]\!]_{\mathbb{Z}_{2^m}/\mathbb{Z}}\}$$

(c) Expressions do not wrap

NO WRAPPING ASSUMPTIONS

Given an expression e with m bits:

$$\mathcal{I}_W(\mathsf{e}) = \{(\underline{i}) \mid [\![\mathsf{e}]\!]_{\mathbb{Z}} \neq [\![\mathsf{e}]\!]_{\mathbb{Z}_{2^m}/\mathbb{Z}}\}$$

Let e be *textually* part of statement S with domain $\mathcal{I}_{\mathsf{S}}$.

(c) Expressions do not wrap

NO WRAPPING ASSUMPTIONS

Given an expression e with m bits:

$$\mathcal{I}_W(\mathsf{e}) = \{(\underline{i}) \mid [\![\mathsf{e}]\!]_{\mathbb{Z}} \neq [\![\mathsf{e}]\!]_{\mathbb{Z}_{2^m}/\mathbb{Z}}\}$$

Let e be *textually* part of statement S with domain $\mathcal{I}_{\mathsf{S}}$.

$$\mathcal{I}_{W_{\mathsf{S}}}(\mathsf{e}) = \mathcal{I}_W(\mathsf{e}) \cap \mathcal{I}_{\mathsf{S}}$$

(c) Expressions do not wrap

No Wrapping Assumptions

Given an expression e with m bits:

$$\mathcal{I}_W(\mathsf{e}) = \{(\underline{i}) \mid [\![\mathsf{e}]\!]_{\mathbb{Z}} \neq [\![\mathsf{e}]\!]_{\mathbb{Z}_{2^m}/\mathbb{Z}}\}$$

Let e be *textually* part of statement S with domain $\mathcal{I}_{\mathsf{S}}$.

$$\mathcal{I}_{W_{\mathsf{S}}}(\mathsf{e}) = \mathcal{I}_W(\mathsf{e}) \cap \mathcal{I}_{\mathsf{S}}$$

$\mathcal{I}_{W_{\mathsf{S}}}(\mathsf{e})$ describes executed *loop instances* for which e *will wrap*.

(c) Expressions do not wrap

NO WRAPPING ASSUMPTIONS

Given an expression **e** with m bits:

$$\mathcal{I}_W(\mathsf{e}) = \{(\underline{i}) \mid [\![\mathsf{e}]\!]_{\mathbb{Z}} \neq [\![\mathsf{e}]\!]_{\mathbb{Z}_{2^m}/\mathbb{Z}}\}$$

Let **e** be *textually* part of statement **S** with domain $\mathcal{I}_{\mathsf{S}}$.

$$\mathcal{I}_{W_{\mathsf{S}}}(\mathsf{e}) = \mathcal{I}_W(\mathsf{e}) \cap \mathcal{I}_{\mathsf{S}}$$

$\mathcal{I}_{W_{\mathsf{S}}}(\mathsf{e})$ describes executed *loop instances* for which **e** *will wrap*.

$\neg\, \mathcal{I}_{W_{\mathsf{S}}}(\mathsf{e})$ describes sufficient *constrains* under which **e** *will not wrap*.

(c) Expressions do not wrap

ASSUMPTION GENERATION

```
double rhs[JMAX][IMAX][5];

assume grid[0] != MAX_VALUE;
for (j = 0; j < grid[0] + 1; j++)
  assume grid[1] != MAX_VALUE;
  for (i = 0; i < grid[1] + 1; i++)
    for (m = 0; m < 5; m++)

      assume &rhs[j][i][m] >= &grid[2] ||
             &rhs[j][i][m + 1] <= &grid[0];
      rhs[j][i][m] = /* ... */;
```

ASSUMPTION GENERATION

```
double rhs[JMAX][IMAX][5];

assume grid[0] != MAX_VALUE;
for (j = 0; j < grid[0] + 1; j++)
  assume grid[1] != MAX_VALUE;
  for (i = 0; i < grid[1] + 1; i++)
    for (m = 0; m < 5; m++)

      assume &rhs[j][i][m] >= &grid[2] ||
             &rhs[j][i][m + 1] <= &grid[0];
      rhs[j][i][m] = /* ... */;
```

ASSUMPTION GENERATION

```
double rhs[JMAX][IMAX][5];

assume grid[0] != MAX_VALUE;
for (j = 0; j < grid[0] + 1; j++)
  assume grid[1] != MAX_VALUE;
  for (i = 0; i < grid[1] + 1; i++)
    for (m = 0; m < 5; m++)

      assume &rhs[j][i][m] >= &grid[2] ||
             &rhs[j][i][m + 1] <= &grid[0];
      rhs[j][i][m] = /* ... */;
```

(d) Accesses stay in-bounds

ASSUMPTION GENERATION

```
double rhs[JMAX][IMAX][5];

assume grid[0] != MAX_VALUE;
for (j = 0; j < grid[0] + 1; j++)
  assume grid[1] != MAX_VALUE;
  for (i = 0; i < grid[1] + 1; i++)
    for (m = 0; m < 5; m++)
      assume j < JMAX && i < IMAX;
      assume &rhs[j][i][m] >= &grid[2] ||
             &rhs[j][i][m + 1] <= &grid[0];
      rhs[j][i][m] = /* ... */;
```

(d) Accesses stay in-bounds

ASSUMPTION HOISTING

```
double rhs[JMAX][IMAX][5];

assume grid[0] != MAX_VALUE;
for (j = 0; j < grid[0] + 1; j++)
  assume grid[1] != MAX_VALUE;
  for (i = 0; i < grid[1] + 1; i++)
    for (m = 0; m < 5; m++)
      assume j < JMAX && i < IMAX;
      assume &rhs[j][i][m] >= &grid[2] ||
             &rhs[j][i][m + 1] <= &grid[0];
      rhs[j][i][m] = /* ... */;
```

ASSUMPTION HOISTING

```
double rhs[JMAX][IMAX][5];

assume grid[0] != MAX_VALUE;
for (j = 0; j < grid[0] + 1; j++)
  assume grid[1] != MAX_VALUE;
  for (i = 0; i < grid[1] + 1; i++)
    for (m = 0; m < 5; m++)
      assume j < JMAX && i < IMAX;
      assume &rhs[j][i][m] >= &grid[2] ||
             &rhs[j][i][m + 1] <= &grid[0];
      rhs[j][i][m] = /* ... */;
```

ASSUMPTION HOISTING

```
double rhs[JMAX][IMAX][5];

assume grid[0] != MAX_VALUE;
for (j = 0; j < grid[0] + 1; j++)
  assume grid[1] != MAX_VALUE;
  for (i = 0; i < grid[1] + 1; i++)
    for (m = 0; m < 5; m++)
      assume j < JMAX && i < IMAX;
      assume &rhs[j][i][m] >= &grid[2] ||
             &rhs[j][i][m + 1] <= &grid[0];
      rhs[j][i][m] = /* ... */;
```

Hoist, Combine & Simplify Assumptions

ASSUMPTION HOISTING

```
double rhs[JMAX][IMAX][5];

assume grid[0] != MAX_VALUE;
for (j = 0; j < grid[0] + 1; j++)
  assume grid[1] != MAX_VALUE;
  for (i = 0; i < grid[1] + 1; i++)
    for (m = 0; m < 5; m++)
      assume j < JMAX && i < IMAX;
      assume &rhs[j][i][m] >= &grid[2] ||
             &rhs[j][i][m + 1] <= &grid[0];
      rhs[j][i][m] = /* ... */;
```

ASSUMPTION HOISTING

```
double rhs[JMAX][IMAX][5];

assume grid[0] != MAX_VALUE;
for (j = 0; j < grid[0] + 1; j++)
  assume grid[1] != MAX_VALUE;
  for (i = 0; i < grid[1] + 1; i++)
    for (m = 0; m < 5; m++)
      assume j < JMAX && i < IMAX;
      assume &rhs[j][i][m] >= &grid[2] ||
             &rhs[j][i][m + 1] <= &grid[0];
      rhs[j][i][m] = /* ... */;
```

Constraints: $0 \leq$ `j` $\leq$ `grid[0]`

Assumption: `j < JMAX`

ASSUMPTION HOISTING

```
double rhs[JMAX][IMAX][5];

assume grid[0] != MAX_VALUE;
for (j = 0; j < grid[0] + 1; j++)
  assume grid[1] != MAX_VALUE;
  for (i = 0; i < grid[1] + 1; i++)
    for (m = 0; m < 5; m++)
      assume j < JMAX && i < IMAX;
      assume &rhs[j][i][m] >= &grid[2] ||
             &rhs[j][i][m + 1] <= &grid[0];
      rhs[j][i][m] = /* ... */;
```

Constraints: $0 \leq j \leq \texttt{grid[0]}$

Assumption: $\texttt{grid[0] < JMAX} \implies \texttt{j < JMAX}$

ASSUMPTION HOISTING

ASSUMPTION HOISTING

Assumptions are Presburger Formulae

ASSUMPTION HOISTING

Assumptions are Presburger Formulae, that can be *analyzed*, *combined* and *transformed*.

ASSUMPTION HOISTING

Assumptions are Presburger Formulae, that can be *analyzed*, *combined* and *transformed*.

Quantifier elimination is used to *eliminate loop variables*.

ASSUMPTION HOISTING

Assumptions are Presburger Formulae, that can be *analyzed*, *combined* and *transformed*.

Quantifier elimination is used to *eliminate loop variables*.

The result is a *pre-condition* of the original assumption.

ASSUMPTION HOISTING

```
double rhs[JMAX][IMAX][5];

assume grid[0] != MAX_VALUE &&
       grid[1] != MAX_VALUE &&
       grid[0] + 1 <= JMAX &&
       grid[1] + 1 <= IMAX &&
       (&rhs[0][0][0] >= &grid[2] ||
        &rhs[grid[0]][grid[1]][5] <= &grid[0]);

for (j = 0; j < grid[0] + 1; j++)
  for (i = 0; i < grid[1] + 1; i++)
    for (m = 0; m < 5; m++)
      rhs[j][i][m] = /* ... */;
```

ASSUMPTION SIMPLIFICATION

ASSUMPTION SIMPLIFICATION

Eliminate Redundant Constraints:

```
assume N < 128 && N < 127;
=>
assume N < 127;
```

Assumption Simplification

Eliminate Redundant Constraints:

```
assume N < 128 && N < 127;
=>
assume N < 127;
```

Approximate Complicated Constraints:

```
assume &B[N + 2 - ((N - 1) % 3)] <= &A[0] ||
       &A[N + 2 - ((N - 1) % 3)] <= &B[0];
```

Assumption Simplification

Eliminate Redundant Constraints:

```
assume N < 128 && N < 127;
=>
assume N < 127;
```

Approximate Complicated Constraints:

```
assume &B[N + 2 - ((N - 1) % 3)] <= &A[0] ||
       &A[N + 2 - ((N - 1) % 3)] <= &B[0];
=>
assume &B[N + 2] <= &A[0] ||
       &A[N + 2] <= &B[0];
```

Evaluation

ASSUMPTION STATISTICS

	SPEC 2006	*SPEC* 2000
No Variant Loads ∧:	553	6
No Aliasing ∧:	132	52
No Wrapping ∧:	611	82
No Out-Of-Bounds ∧:	5	6
No Unbounded Loop ∧:	42	6
Total:	1343	152

Assumption Statistics

	SPEC 2006	*SPEC* 2000
No Variant Loads $\wedge$:	553	6
No Aliasing $\wedge$:	132	52
No Wrapping $\wedge$:	611	82
No Out-Of-Bounds $\wedge$:	5	6
No Unbounded Loop $\wedge$:	42	6
Total:	1343	152
After Simplification:	$<$ 671 (or $<$ 50%)	$<$ 99 (or $<$ 66%)

Assumption Statistics

	SPEC 2006	*SPEC 2000*
No Variant Loads Λ:	553	6
No Aliasin		52
No Wrappin		82
No Out-Of-Bound		6
No Unbounded Loo		6
To		152
After Simplification:	< 671 (or $< 50\%$)	< 99 (or $< 66\%$)

Two's complement modeling increased compile time by $3 - 3000\%$.

Applicability & Validity

SPEC 2006

	w/o Λssumptions	w/ Λssumptions	
modeled:	35	191	×5.45
feasible:	35	102	×2.91
executed:	61k	5.2M	×85.24
valid:	61k	99.68% ∗ 5.2M	×85.21

SPEC 2000

	w/o Λssumptions	w/ Λssumptions	
modeled:	24	83	×3.45
feasible:	24	78	×3.25
executed:	11k	729k	×66.27
valid:	11k	89.3% ∗ 729k	×59.18

APPLICABILITY & VALIDITY

SPEC 2006

	w/o Assumptions	w/ Assumptions	
modeled:		191	×5.45
feasible:		102	×2.91
executed:		5.2M	×85.24
valid:		5.2M	×85.21

Assumptions fail

$\approx 2\%$

of the time and cause

$< 4\%$

runtime overhead.

		tions	
modeled:	24	83	×3.45
feasible:	24	78	×3.25
executed:	11k	729k	×66.27
valid:	11k	$89.3\% * 729k$	×59.18

Conclusion

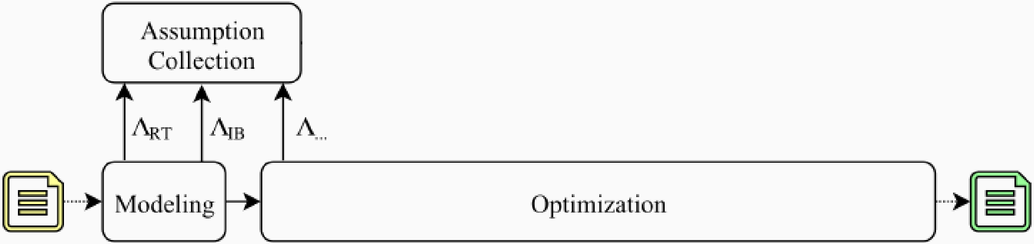
ARCHITECTURE OVERVIEW

ARCHITECTURE OVERVIEW

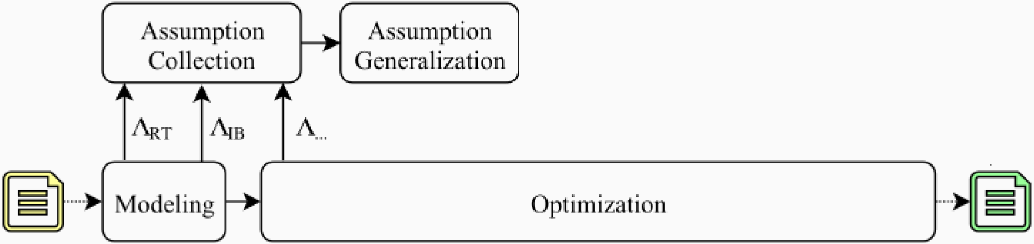
Modeling

Optimization

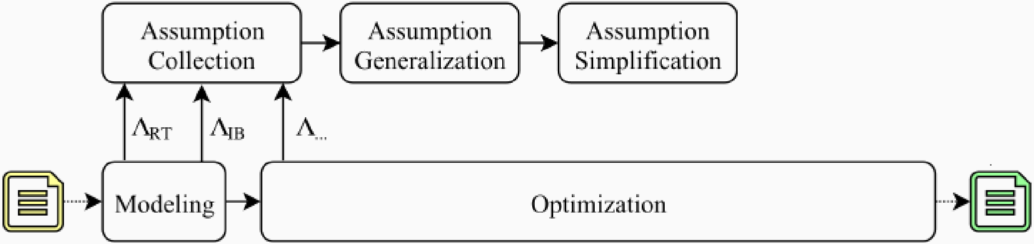
ARCHITECTURE OVERVIEW



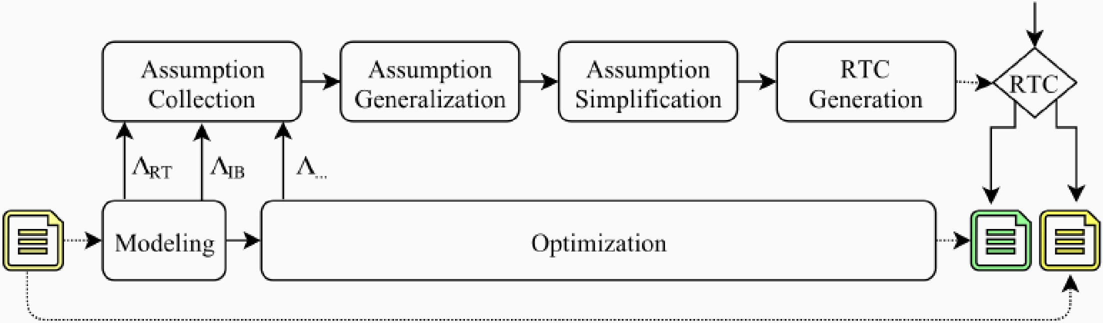
ARCHITECTURE OVERVIEW



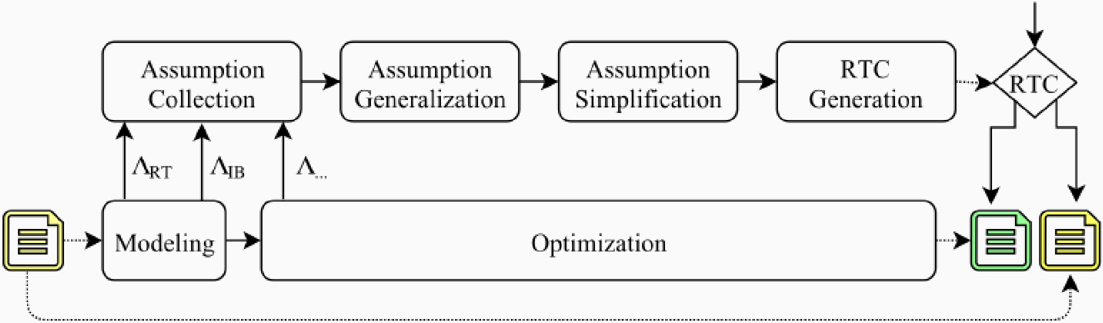
Architecture Overview



ARCHITECTURE OVERVIEW



ARCHITECTURE OVERVIEW



Thank You.

Backup

FINITE LOOP ASSUMPTION

Infinite loops create unbounded optimization problems

FINITE LOOP ASSUMPTION

Infinite loops create unbounded optimization problems

```
for (unsigned i = 0; i != N; i+=2)
  A[i+4] = A[i];
```

FINITE LOOP ASSUMPTION

Infinite loops create unbounded optimization problems

```
if (N % 2 == 0) {

  for (unsigned i = 0; i != N; i+=2)
    A[i+4] = A[i];

} else {
  /* original code */
}
```

INVARIANT LOAD ASSUMPTIONS

```
for (i = 0; i < *Size1; i++)
  for (j = 0; j < *Size0; j++)
    ...
```

INVARIANT LOAD ASSUMPTIONS

```
auto Size0V, Size1V = *Size1;

if (Size1V > 0)
  Size0V = *Size0;

for (i = 0; i < Size1V; i++)
  for (j = 0; j < Size0V; j++)
    ...
```

Hoist invariant loads but *keep control conditions* intact.

INVARIANT LOAD ASSUMPTIONS

```
auto Size0V, Size1V = *Size1;

if (Size1V > 0)
  Size0V = *Size0;

for (i = 0; i < Size1V; i++)
  for (j = 0; j < Size0V; j++)
    ...
```

Hoist invariant loads but *keep control conditions* intact.

Powerful *in combination* with *runtime alias checks*.

ASSUMPTION SIMPLIFICATION

Simplify Complicated Constraints:

```
assume &B[N + 2 - ((N - 1) % 3)] <= &A[0] ||
       &A[N + 2 - ((N - 1) % 3)] <= &B[0];

assume &B[N + 2] <= &A[0] ||
       &A[N + 2] <= &B[0];

for (i = 0; i < N; i += 3) {
  A[i + 0] += 1.3 * B[i + 0];
  A[i + 1] += 1.7 * B[i + 1];
  A[i + 2] += 2.1 * B[i + 2];
}
```

Sound & Automatic Polyhedral Optimization

Polyhedral optimizations show *great performance improvements*,

SOUND & AUTOMATIC POLYHEDRAL OPTIMIZATION

Polyhedral optimizations show *great performance improvements*,
though they often require *manual pre-processing*
and are *unsound* for corner case inputs.

SOUND & AUTOMATIC POLYHEDRAL OPTIMIZATION

Polyhedral optimizations show *great performance improvements*,
though they often require *manual pre-processing*
and are *unsound* for corner case inputs.

SPEC 2006 – 456.hmmer – P7_Viterbi
−28% execution time

NAS Parallel Benchmark Suite – BT – compute_rhs
6× fold speedup with 8 threads [Metha and Yew, PLDI'15]

Semantic Differences

	Rust	Java	C	LLVM-IR	Polyhedral Model
Variant Loads in Control Conditions	✓	✓	✓	✓	✗
Aliasing Arrays	✗	✗	✓	✓	✗
Integer Wrapping	✓	✓	✓	✓	✗
Out-of-Bound Accesses	✓	✓	✓	✓	✗
Potentially Unbounded Loops	✓	✓	✓	✓	✗