Motivating Example
A Potentially Parallel Loop

```c
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 (R)
{ A[N + i] | 0 ≤ i < N }

Write Set (W)
{ A[i] | 0 ≤ i < N }
A POTENTIALLY PARALLEL LOOP

\[\text{for } (i = 0; i < N; i++) \]

Read Set (R)
\[\{ A[N + i] | \theta \leq i < N \}\]

Write Set (W)
\[\{ A[i] | \theta \leq i < N \}\]

\[R \cap W = \{ \}\]
A Potentially Parallel Loop

\[
\text{for } (i = 0; i < N; i++) \\
\]

Read Set (R) \hspace{1cm} Write Set (W)
\[
\{ A[N + i] | 0 \leq i < N \} \hspace{1cm} \{ A[i] | 0 \leq i < N \}
\]

\[ R \cap W = \{ \} \hspace{1cm} \text{Parallel} \]
unsigned char i, N;

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

Read Set (R)
{ A[N + i] | 0 ≤ i < N }

Write Set (W)
{ A[i] | 0 ≤ i < N }
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 \( (W) \)  
\( \{ A[i] \mid 0 \leq i < N \} \)
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 \} \quad \text{Write Set (W)}
\{ A[i] \mid 0 \leq i < N \}
\{ A[(N + i) \text{ mod } 256] \mid \ldots \}
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
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)
\(\{ A[i] \mid 0 \leq i < N \}\)

\(\{ A[(N + i) \mod 256] \mid ... \}\)

\(R \cap W = \{ \}\), iff \(N \leq 128\)  

Potentially Sequential
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
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

/* loop nest */
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

```c
if ( )
    /* optimized loop nest */
else
    /* loop nest */
```
1. Take *Optimistic Assumptions* to model the loop nest
2. Optimize the loop nest
3. Version the code
4. Create a *simple* runtime check

```c
if (/* simple runtime check */)
    /* optimized loop nest */
else
    /* loop nest */
```
<table>
<thead>
<tr>
<th></th>
<th>C</th>
<th>LLVM-IR</th>
<th>Polyhedral Model</th>
</tr>
</thead>
</table>

### Semantic Differences

<table>
<thead>
<tr>
<th>C</th>
<th>LLVM-IR</th>
<th>Polyhedral Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variant Loads in Control Conditions</td>
<td>✔</td>
<td>✔</td>
</tr>
</tbody>
</table>
## Semantic Differences

<table>
<thead>
<tr>
<th></th>
<th>C</th>
<th>LLVM-IR</th>
<th>Polyhedral Model</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Variant Loads in Control Conditions</strong></td>
<td>✔️</td>
<td>✔️</td>
<td>✗</td>
</tr>
<tr>
<td><strong>Aliasing Arrays</strong></td>
<td>✔️</td>
<td>✔️</td>
<td>✗</td>
</tr>
</tbody>
</table>
## Semantic Differences

<table>
<thead>
<tr>
<th></th>
<th>C</th>
<th>LLVM-IR</th>
<th>Polyhedral Model</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Variant Loads in Control Conditions</strong></td>
<td>✔</td>
<td>✔</td>
<td>✗</td>
</tr>
<tr>
<td><strong>Aliasing Arrays</strong></td>
<td>✔</td>
<td>✔</td>
<td>✗</td>
</tr>
<tr>
<td><strong>Integer Wrapping</strong></td>
<td>✔</td>
<td>✔</td>
<td>✗</td>
</tr>
</tbody>
</table>
## Semantic Differences

<table>
<thead>
<tr>
<th></th>
<th>C</th>
<th>LLVM-IR</th>
<th>Polyhedral Model</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Variant Loads in Control Conditions</strong></td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td><strong>Aliasing Arrays</strong></td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td><strong>Integer Wrapping</strong></td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td><strong>Out-of-Bound Accesses</strong></td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
</tr>
</tbody>
</table>
## Semantic Differences

<table>
<thead>
<tr>
<th></th>
<th>C</th>
<th>LLVM-IR</th>
<th>Polyhedral Model</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Variant Loads in Control Conditions</strong></td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td><strong>Aliasing Arrays</strong></td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td><strong>Integer Wrapping</strong></td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td><strong>Out-of-Bound Accesses</strong></td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td><strong>Potentially Unbounded Loops</strong></td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
</tr>
</tbody>
</table>
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
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] = /* ... */;
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
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
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
double \text{rhs}[JMAX][IMAX][5];

\text{assume} \quad \text{grid}[0] \neq \text{MAX\_VALUE};
\text{for} \quad (j = 0; j < \text{grid}[0] + 1; j++)
\quad \text{assume} \quad \text{grid}[1] \neq \text{MAX\_VALUE};
\text{for} \quad (i = 0; i < \text{grid}[1] + 1; i++)
\text{for} \quad (m = 0; m < 5; m++)

\text{assume} \quad \& \text{rhs}[j][i][m] \geq \& \text{grid}[2] \quad || \quad
\& \text{rhs}[j][i][m + 1] \leq \& \text{grid}[0];
\text{rhs}[j][i][m] = \texttt{/* ... */};

(c) Expressions do not wrap
Given an expression $e$ with $m$ bits:

(c) Expressions do not wrap
Given an expression $e$ with $m$ bits:

$$[e]_z$$

(c) Expressions do not wrap
No Wrapping Assumptions

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

$$[e]_Z \quad [e]_{Z^{2m}/Z}$$

$(c)$ Expressions do not wrap
No Wrapping Assumptions

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

$$[e]_Z \neq [e]_{Z^{2m}/Z}$$

(c) Expressions do not wrap
Given an expression \( e \) with \( m \) bits:

\[
\mathcal{I}_W(e) = \{(i) \mid \llbracket e \rrbracket_z \neq \llbracket e \rrbracket_{z^{2m}/z}\}
\]
Given an expression $e$ with $m$ bits:

$$\mathcal{I}_W(e) = \{(i) \mid [e]_z \neq [e]_{2^m/z}\}$$

Let $e$ be \textit{textually} part of statement $S$ with domain $\mathcal{I}_S$. 

\textbf{(c) Expressions do not wrap}
No Wrapping Assumptions

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

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

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

$$\mathcal{I}_{W_S}(e) = \mathcal{I}_W(e) \cap \mathcal{I}_S$$

(c) Expressions do not wrap
Given an expression $e$ with $m$ bits:

$$\mathcal{I}_W(e) = \{(i) \mid \lfloor e \rfloor_z \neq \lfloor e \rfloor_{2^m/z}\}$$

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

$$\mathcal{I}_{W_S}(e) = \mathcal{I}_W(e) \cap \mathcal{I}_S$$

$\mathcal{I}_{W_S}(e)$ describes executed loop instances for which $e$ will wrap.

---

(c) Expressions do not wrap
NO WRAPPING ASSUMPTIONS

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

$$I_W(e) = \{(i) \mid \! [e]_Z \neq \! [e]_{Z^{2^m}} \}$$

Let $e$ be textually part of statement $S$ with domain $I_S$.

$$I_W_S(e) = I_W(e) \cap I_S$$

$I_W_S(e)$ describes executed loop instances for which $e$ will wrap.

$\neg I_W_S(e)$ describes sufficient constrains under which $e$ will not wrap.

\( (c) \) Expressions do not wrap
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] = /* ... */;
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
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**

```c
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] = /* ... */;
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] = /* ... */;
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] = /* ... */;
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] = /* ... */;
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 ≤ j ≤ grid[0]
Assumption: j < JMAX
double \( \text{rhs}[\text{JMAX}][\text{IMAX}][5] \);

\textbf{assume} \quad \text{grid}[0] \neq \text{MAX\_VALUE} ;

\textbf{for} ( j = 0; j < \text{grid}[0] + 1; j++ )

\textbf{assume} \quad \text{grid}[1] \neq \text{MAX\_VALUE} ;

\textbf{for} ( i = 0; i < \text{grid}[1] + 1; i++ )

\textbf{for} ( m = 0; m < 5; m++ )

\begin{itemize}
  \item \textbf{assume} \quad j < \text{JMAX} \quad \&\& \quad i < \text{IMAX} ;
  \item \textbf{assume} \quad \text{rhs}[j][i][m] \geq \text{grid}[2] \quad \| \\
  \quad \text{rhs}[j][i][m + 1] \leq \text{grid}[0] ;
  \item \text{rhs}[j][i][m] = */ \ldots */ ;
\end{itemize}

\begin{flushright}
\textbf{Constraints:} \quad 0 \leq j \leq \text{grid}[0] \\
\textbf{Assumption:} \quad \text{grid}[0] < \text{JMAX} \implies j < \text{JMAX}
\end{flushright}
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

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

Eliminate Redundant Constraints:

\[
\text{assume } N < 128 \land N < 127;
\]
\[
\Rightarrow
\]
\[
\text{assume } N < 127;
\]
Assumption Simplification

Eliminate Redundant Constraints:

\[\text{assume } N < 128 \land N < 127; \]
\[\Rightarrow\]
\[\text{assume } N < 127;\]

Approximate Complicated Constraints:

\[\text{assume } \&B[N + 2 - ((N - 1) \mod 3)] \leq \&A[0] \lor \]
\[\&A[N + 2 - ((N - 1) \mod 3)] \leq \&B[0];\]
Assumption Simplification

Eliminate Redundant Constraints:

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

Approximate Complicated Constraints:

```plaintext
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

<table>
<thead>
<tr>
<th>Assumption</th>
<th>SPEC 2006</th>
<th>SPEC 2000</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Variant Loads Λ:</td>
<td>553</td>
<td>6</td>
</tr>
<tr>
<td>No Aliasing Λ:</td>
<td>132</td>
<td>52</td>
</tr>
<tr>
<td>No Wrapping Λ:</td>
<td>611</td>
<td>82</td>
</tr>
<tr>
<td>No Out-Of-Bounds Λ:</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>No Unbounded Loop Λ:</td>
<td>42</td>
<td>6</td>
</tr>
<tr>
<td><strong>Total:</strong></td>
<td><strong>1343</strong></td>
<td><strong>152</strong></td>
</tr>
</tbody>
</table>
## Assumption Statistics

<table>
<thead>
<tr>
<th>Assumption</th>
<th>SPEC 2006</th>
<th>SPEC 2000</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Variant Loads ( \Lambda ):</td>
<td>553</td>
<td>6</td>
</tr>
<tr>
<td>No Aliasing ( \Lambda ):</td>
<td>132</td>
<td>52</td>
</tr>
<tr>
<td>No Wrapping ( \Lambda ):</td>
<td>611</td>
<td>82</td>
</tr>
<tr>
<td>No Out-Of-Bounds ( \Lambda ):</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>No Unbounded Loop ( \Lambda ):</td>
<td>42</td>
<td>6</td>
</tr>
<tr>
<td><strong>Total:</strong></td>
<td><strong>1343</strong></td>
<td><strong>152</strong></td>
</tr>
<tr>
<td>After Simplification:</td>
<td>&lt; 671 (or &lt; 50%)</td>
<td>&lt; 99 (or &lt; 66%)</td>
</tr>
<tr>
<td>Assumption</td>
<td>SPEC 2006</td>
<td>SPEC 2000</td>
</tr>
<tr>
<td>-----------------------------------</td>
<td>-----------</td>
<td>-----------</td>
</tr>
<tr>
<td>No Variant Loads Λ:</td>
<td>553</td>
<td>6</td>
</tr>
<tr>
<td>No Aliasing:</td>
<td>122</td>
<td>52</td>
</tr>
<tr>
<td>No Wrapping:</td>
<td>611</td>
<td>82</td>
</tr>
<tr>
<td>No Out-Of-Bounds:</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>No Unbounded Loops:</td>
<td>42</td>
<td>6</td>
</tr>
<tr>
<td>Total</td>
<td>1343</td>
<td>152</td>
</tr>
</tbody>
</table>

After Simplification:

- < 671 (or < 50%)
- < 99 (or < 66%)

Two’s complement modeling increased compile time by 3 – 3000%.
## Applicability & Validity

### SPEC 2006

<table>
<thead>
<tr>
<th></th>
<th>w/o Assumptions</th>
<th>w/ Assumptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>modeled:</td>
<td>35</td>
<td>191</td>
</tr>
<tr>
<td>feasible:</td>
<td>35</td>
<td>102</td>
</tr>
<tr>
<td>executed:</td>
<td>61k</td>
<td>5.2M</td>
</tr>
<tr>
<td>valid:</td>
<td>61k</td>
<td>99.68% * 5.2M</td>
</tr>
</tbody>
</table>

### SPEC 2000

<table>
<thead>
<tr>
<th></th>
<th>w/o Assumptions</th>
<th>w/ Assumptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>modeled:</td>
<td>24</td>
<td>83</td>
</tr>
<tr>
<td>feasible:</td>
<td>24</td>
<td>78</td>
</tr>
<tr>
<td>executed:</td>
<td>11k</td>
<td>729k</td>
</tr>
<tr>
<td>valid:</td>
<td>11k</td>
<td>89.3% * 729k</td>
</tr>
</tbody>
</table>

× 5.45
× 2.91
× 85.24
× 85.21
× 3.45
× 3.25
× 66.27
× 59.18
<table>
<thead>
<tr>
<th></th>
<th>SPEC 2006 w/o Assumptions</th>
<th>SPEC 2006 w/ Assumptions</th>
</tr>
</thead>
<tbody>
<tr>
<td>Modeled:</td>
<td>191 (× 5.45)</td>
<td>83 (× 3.45)</td>
</tr>
<tr>
<td>Feasible:</td>
<td>102 (× 2.91)</td>
<td>78 (× 3.25)</td>
</tr>
<tr>
<td>Executed:</td>
<td>5.2M (× 85.24)</td>
<td>729k (× 66.27)</td>
</tr>
<tr>
<td>Valid:</td>
<td>5.2M (× 85.21)</td>
<td>89.3% * 729k (× 59.18)</td>
</tr>
</tbody>
</table>

Assumptions fail approximately 2% of the time and cause less than 4% runtime overhead.
Conclusion
Architecture Overview

Modeling → Optimization
Thank You.
Backup
Infinite loops create unbounded optimization problems
Finite Loop Assumption

Infinite loops create unbounded optimization problems

```c
for (unsigned i = 0; i != N; i += 2)
    A[i+4] = A[i];
```
Infinite loops create unbounded optimization problems

```c
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++)
    ...

Hoist invariant loads but keep control conditions intact.
Powerful in combination with runtime alias checks.
**Invariant Load Assumptions**

```c++
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.
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:

\[
\text{assume } \&B[N + 2 - ((N - 1) \% 3)] \leq \&A[0] \mid \mid \\
\&A[N + 2 - ((N - 1) \% 3)] \leq \&B[0];
\]

\[
\text{assume } \&B[N + 2] \leq \&A[0] \mid \mid \\
\&A[N + 2] \leq \&B[0];
\]

for \((i = 0; i < N; i += 3)\) {
    \(A[i + 0] += 1.3 * B[i + 0];\)
    \(A[i + 1] += 1.7 * B[i + 1];\)
    \(A[i + 2] += 2.1 * B[i + 2];\)
}
Polyhedral optimizations show great performance improvements,
Polyhedral optimizations show great performance improvements, though they often require manual pre-processing and are unsound for corner case inputs.
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

<table>
<thead>
<tr>
<th></th>
<th>Rust</th>
<th>Java</th>
<th>C</th>
<th>LLVM-IR</th>
<th>Polyhedral Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variant Loads in Control Conditions</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>Aliasing Arrays</td>
<td>✗</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>Integer Wrapping</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>Out-of-Bound Accesses</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>Potentially Unbounded Loops</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
</tr>
</tbody>
</table>