Experience with Widening based Equivalence Checking in Realistic Multimedia Systems

Sven Verdoolaege Martin Palkovic Maurice Bruynooghe
Gerda Janssens Francky Catthoor

Katholieke Universiteit Leuven, Belgium

IMEC, Leuven, Belgium

November 6, 2009

Equivalence Checking Example

```
for (i = 0; i < M; ++i) {
    m = i+1;
    for (j = 0; j < N; ++j)
        m = g(h(m), in1[i][j]);
    compute_row(h(m), A[M-i-1]);
}
A[5][6] = 0;
for (i = 0; i < M - 2; ++i)
    out[i] = f(A[M-i-2-in2[i]]);

for (i = 0; i < M; ++i) {
    m = h(i+1);
    for (j = 0; j < N; ++j)
        m = h(g(m, in1[i][j]));
    compute_row(m, B[i]);
    if (i >= 2)
        out[i-2] = f(B[i-1+in2[i-2]]);
}
```

Are the two programs on the left equivalent?

⇒ Same **output** when given same **input**

Yes, except at $[M-8, M-6]$ (when value of `in2` in $[-1,1]$)

Assumptions

- no pointers
- no recursion
- functions called are pure
- static control flow
- quasi-affine loop bounds
- quasi-affine conditions
- quasi-affine index expressions

Equivalence Checking Example

```
for (i = 0; i < M; ++i) {
    m = i+1;
    for (j = 0; j < N; ++j)
        m = g(h(m), in1[i][j]);
    compute_row(h(m), A[M-i-1]);
}
A[5][6] = 0;
for (i = 0; i < M - 2; ++i)
    out[i] = f(A[M-i-2-in2[i]]);

for (i = 0; i < M; ++i) {
    m = h(i+1);
    for (j = 0; j < N; ++j)
        m = h(g(m, in1[i][j]));
    compute_row(m, B[i]);
    if (i >= 2)
        out[i-2] = f(B[i-1+in2[i-2]]);
}
```

Are the two programs on the left equivalent?

⇒ Same output when given same input

Yes, except at $[M-8, M-6]$ (when value of $in2$ in $[-1,1]$)

Assumptions

- no pointers
- no recursion
- functions called are pure
- static control flow
- quasi-affine loop bounds
- quasi-affine conditions
- quasi-affine index expressions

Supported Constructs:

- Parameters
- Recurrences
- Row accesses
- Data-dependent reads

Equivalence Checking Example

```
for (i = 0; i < M; ++i) {  
    m = i+1;  
    for (j = 0; j < N; ++j)  
        m = g(h(m), in1[i][j]);  
    compute_row(h(m), A[M-i-1]);  
}  
A[5][6] = 0;  
for (i = 0; i < M - 2; ++i)  
    out[i] = f(A[M-i-2-in2[i]]);  
for (i = 0; i < M; ++i) {  
    m = h(i+1);  
    for (j = 0; j < N; ++j)  
        m = h(g(m, in1[i][j]));  
    compute_row(m, B[i]);  
    if (i >= 2)  
        out[i-2] = f(B[i-1+in2[i-2]]);  
}
```

Are the two programs on the left equivalent?

⇒ Same output when given same input

Yes, except at $[M-8, M-6]$ (when value of $in2$ in $[-1,1]$)

Assumptions

- no pointers
- no recursion
- functions called are pure
- static control flow
- quasi-affine loop bounds
- quasi-affine conditions
- quasi-affine index expressions

Supported Constructs:

- Parameters
- Recurrences
- Row accesses
- Data-dependent reads

Equivalence Checking Example

```
for (i = 0; i < M; ++i) {
    m = i+1;
    for (j = 0; j < N; ++j)
        m = g(h(m), in1[i][j]);
    compute_row(h(m), A[M-i-1]);
}
A[5][6] = 0;
for (i = 0; i < M - 2; ++i)
    out[i] = f(A[M-i-2-in2[i]]);

for (i = 0; i < M; ++i) {
    m = h(i+1);
    for (j = 0; j < N; ++j)
        m = h(g(m, in1[i][j]));
    compute_row(m, B[i]);
    if (i >= 2)
        out[i-2] = f(B[i-1+in2[i-2]]);
}
```

Are the two programs on the left equivalent?

⇒ Same output when given same input

Yes, except at $[M-8, M-6]$ (when value of $in2$ in $[-1,1]$)

Assumptions

- no pointers
- no recursion
- functions called are pure
- static control flow
- quasi-affine loop bounds
- quasi-affine conditions
- quasi-affine index expressions

Supported Constructs:

- Parameters
- **Recurrences**
- Row accesses
- Data-dependent reads

Equivalence Checking Example

```
for (i = 0; i < M; ++i) {  
    m = i+1;  
    for (j = 0; j < N; ++j)  
        m = g(h(m), in1[i][j]);  
    compute_row(h(m), A[M-i-1]);  
}  
A[5][6] = 0;  
for (i = 0; i < M - 2; ++i)  
    out[i] = f(A[M-i-2-in2[i]]);  
  
for (i = 0; i < M; ++i) {  
    m = h(i+1);  
    for (j = 0; j < N; ++j)  
        m = h(g(m, in1[i][j]));  
    compute_row(m, B[i]);  
    if (i >= 2)  
        out[i-2] = f(B[i-1+in2[i-2]]);  
}
```

Are the two programs on the left equivalent?

⇒ Same output when given same input

Yes, except at $[M-8, M-6]$ (when value of $in2$ in $[-1,1]$)

Assumptions

- no pointers
- no recursion
- functions called are pure
- static control flow
- quasi-affine loop bounds
- quasi-affine conditions
- quasi-affine index expressions

Supported Constructs:

- Parameters
- Recurrences
- Row accesses
- Data-dependent reads

Equivalence Checking Example

```
for (i = 0; i < M; ++i) {
    m = i+1;
    for (j = 0; j < N; ++j)
        m = g(h(m), in1[i][j]);
    compute_row(h(m), A[M-i-1]);
}
A[5][6] = 0;
for (i = 0; i < M - 2; ++i)
    out[i] = f(A[M-i-2-in2[i]]);

for (i = 0; i < M; ++i) {
    m = h(i+1);
    for (j = 0; j < N; ++j)
        m = h(g(m, in1[i][j]));
    compute_row(m, B[i]);
    if (i >= 2)
        out[i-2] = f(B[i-1+in2[i-2]]);
}
```

Are the two programs on the left equivalent?

⇒ Same output when given same input

Yes, except at $[M-8, M-6]$ (when value of $in2$ in $[-1,1]$)

Assumptions

- no pointers
- no recursion
- functions called are pure
- static control flow
- quasi-affine loop bounds
- quasi-affine conditions
- quasi-affine index expressions

Supported Constructs:

- Parameters
- Recurrences
- Row accesses
- **Data-dependent reads**

Context

Multi-media applications for embedded systems

- Manipulation of large multi-dimensional arrays
- Relatively simple control flow, mainly loops
- Strict timing constraints

Transformation are applied

- mainly loop transformations and data transformations
- to improve time and energy efficiency
- (semi)automatically or manually

Question:

Is the transformed program (input-output) equivalent to the original?

⇒ equivalence checking should be fully automated

Outline

- 1 Motivation
- 2 Basic Approach
 - Dependence Graphs
 - Propagation
- 3 Extensions
 - Motivation
 - Expansion Edges
 - Data-dependent Reads
 - Row accesses
- 4 Implementation
- 5 Experimental Results
- 6 Conclusion

Basic Proof Strategy

Prove output arrays are equal given that input arrays are equal

- ① Compute weakest precondition for equal output
 - ⇒ propagate correspondences between values computed in both programs
- ② Check which part is covered by equal input
- ③ Propagate results back to output
 - ⇒ which elements of the output arrays are equal
 - ⇒ recurrence handling
 - ⇒ commutativity handling

All functions are assumed to be pure and are treated as black-boxes

⇒ results of function calls equal iff all arguments are equal

Equivalence checking performed on *dependence graph* abstraction

Dependence Graphs

Dependence graph contains the following information:
for each value computed in the program, on which values does it depend?

Each computed value is represented by

- function call/operation used to compute value
- iteration vector of the enclosing loops

⇒ complete abstraction of where/how values are stored in memory

Dependence Graphs

Dependence graph contains the following information:
for each value computed in the program, on which values does it depend?

Each computed value is represented by

- function call/operation used to compute value
- iteration vector of the enclosing loops

⇒ complete abstraction of where/how values are stored in memory

Dependence graph

- vertices: “computations”
 - ▶ a (static) function call or operation
 - ▶ iteration domain D : set of iterations of enclosing loops
- edges: dependences
 - ▶ argument position p
 - ▶ dependence relation M : maps use to (unique) definition

Dependences are obtained through exact dataflow analysis

Dependence Graphs: Trivial Example

```
for (i = 0; i <= N; ++i)
    a[i] = in[i];
a[0] = in[1];
for (i = 0; i <= N; ++i)
    b[i] = f(a[N-i]);
for (i = 0; i <= N; ++i)
    Out(b[i]);
```

Dependence graph

- vertices: “computations”
 - ▶ a (static) function call or operation
 - ▶ iteration domain D : set of iterations of enclosing loops
- edges: dependences
 - ▶ argument position p
 - ▶ dependence relation M : maps use to (unique) definition

Dependences are obtained through exact dataflow analysis

Dependence Graphs: Trivial Example

```
for (i = 0; i <= N; ++i)
    a[i] = in[i];
a[0] = in[1];
for (i = 0; i <= N; ++i)
    b[i] = f(a[N-i]);
for (i = 0; i <= N; ++i)
    Out(b[i]);
```

Dependence graph

- vertices: “computations”
 - ▶ a (static) function call or operation
 - ▶ iteration domain D : set of iterations of enclosing loops
- edges: dependences
 - ▶ argument position p
 - ▶ dependence relation M : maps use to (unique) definition

Dependences are obtained through exact dataflow analysis

Dependence Graphs: Trivial Example

```
for (i = 0; i <= N; ++i)
  a[i] = in[i];
a[0] = in[1];
for (i = 0; i <= N; ++i)
  b[i] = f(a[N-i]);
for (i = 0; i <= N; ++i)
  Out(b[i]);
```

Dependence graph

- vertices: “computations”
 - ▶ a (static) function call or operation
 - ▶ iteration domain D : set of iterations of enclosing loops
- edges: dependences
 - ▶ argument position p
 - ▶ dependence relation M : maps use to (unique) definition

Dependences are obtained through exact dataflow analysis

Dependence Graphs: Trivial Example

```
for (i = 0; i <= N; ++i)
    a[i] = in[i];
a[0] = in[1];
for (i = 0; i <= N; ++i)
    b[i] = f(a[N-i]);
for (i = 0; i <= N; ++i)
    Out(b[i]);
```

Dependence graph

- vertices: “computations”
 - ▶ a (static) function call or operation
 - ▶ iteration domain D : set of iterations of enclosing loops
- edges: dependences
 - ▶ argument position p
 - ▶ dependence relation M : maps use to (unique) definition

Dependences are obtained through exact dataflow analysis

Dependence Graphs: Trivial Example

```

for (i = 0; i <= N; ++i)
    a[i] = in[i];
a[0] = in[1];
for (i = 0; i <= N; ++i)
    b[i] = f(a[N-i]);
for (i = 0; i <= N; ++i)
    Out(b[i]);

```

```

for (i = 0; i <= N; ++i) {
    a[i] = in[i];
    b[i] = f(a[i])
}
for (i = 0; i <= N; ++i)
    Out(b[N-i]);

```

Propagation of Correspondences

Propagation of Correspondences

Initial correspondence: $R_{e,w}^{want} = \{i \leftrightarrow i \mid 0 \leq i \leq N\}$

Propagation of Correspondences

Initial correspondence: $R_{e,w}^{want} = \{i \leftrightarrow i \mid 0 \leq i \leq N\}$

Propagation: $R_{d,v}^{want} = (M_{(e,d)} \oplus M_{(w,v)}) R_{e,w}^{want}$

Propagation of Correspondences

Initial correspondence: $R_{e,w}^{want} = \{i \leftrightarrow i \mid 0 \leq i \leq N\}$

Propagation: $R_{d,v}^{want} = (M_{(e,d)} \oplus M_{(w,v)}) R_{e,w}^{want}$

Propagation of Correspondences

Initial correspondence: $R_{e,w}^{want} = \{i \leftrightarrow i \mid 0 \leq i \leq N\}$

Propagation: $R_{d,v}^{want} = (M_{(e,d)} \oplus M_{(w,v)}) R_{e,w}^{want}$

Propagation of Correspondences

Initial correspondence: $R_{e,w}^{want} = \{i \leftrightarrow i \mid 0 \leq i \leq N\}$

Propagation: $R_{d,v}^{want} = (M_{(e,d)} \oplus M_{(w,v)}) R_{e,w}^{want}$

Correspondence at input nodes holds for equal indices

$$\Rightarrow R^{got} = R^{want} \cap \{i \leftrightarrow i\}$$

$$\Rightarrow R^{lost} = R^{want} \setminus \{i \leftrightarrow i\}$$

Propagation of Correspondences

Initial correspondence: $R_{e,w}^{want} = \{i \leftrightarrow i \mid 0 \leq i \leq N\}$

Propagation: $R_{d,v}^{want} = (M_{(e,d)} \oplus M_{(w,v)}) R_{e,w}^{want}$

Correspondence at input nodes holds for equal indices

$$\Rightarrow R^{got} = R^{want} \cap \{i \leftrightarrow i\}$$

$$\Rightarrow R^{lost} = R^{want} \setminus \{i \leftrightarrow i\}$$

Propagation of Correspondences

Initial correspondence: $R_{e,w}^{want} = \{i \leftrightarrow i \mid 0 \leq i \leq N\}$

Propagation: $R_{d,v}^{want} = (M_{(e,d)} \oplus M_{(w,v)}) R_{e,w}^{want}$

Correspondence at input nodes holds for equal indices

$$\Rightarrow R^{got} = R^{want} \cap \{i \leftrightarrow i\}$$

$$\Rightarrow R^{lost} = R^{want} \setminus \{i \leftrightarrow i\}$$

Backpropagation

$$\bullet R_{u_1, u_2}^{lost} = \bigcup_{(u_1, v_1), (u_2, v_2) \text{ s.t. } p_{(u_1, v_1)} = p_{(u_2, v_2)}} \left(M_{(u_1, v_1)}^{-1} \oplus M_{(u_2, v_2)}^{-1} \right) R_{v_1, v_2}^{lost}$$

$$\bullet R^{got} = R^{want} \setminus R^{lost}$$

Propagation of Correspondences

Initial correspondence: $R_{e,w}^{want} = \{i \leftrightarrow i \mid 0 \leq i \leq N\}$

Propagation: $R_{d,v}^{want} = (M_{(e,d)} \oplus M_{(w,v)}) R_{e,w}^{want}$

Correspondence at input nodes holds for equal indices

$$\Rightarrow R^{got} = R^{want} \cap \{i \leftrightarrow i\}$$

$$\Rightarrow R^{lost} = R^{want} \setminus \{i \leftrightarrow i\}$$

Backpropagation

$$\bullet R_{u_1, u_2}^{lost} = \bigcup_{(u_1, v_1), (u_2, v_2) \text{ s.t. } p_{(u_1, v_1)} = p_{(u_2, v_2)}} \left(M_{(u_1, v_1)}^{-1} \oplus M_{(u_2, v_2)}^{-1} \right) R_{v_1, v_2}^{lost}$$

$$\bullet R^{got} = R^{want} \setminus R^{lost}$$

Propagation of Correspondences

Initial correspondence: $R_{e,w}^{\text{want}} = \{i \leftrightarrow i \mid 0 \leq i \leq N\}$

Propagation: $R_{d,v}^{\text{want}} = (M_{(e,d)} \oplus M_{(w,v)}) R_{e,w}^{\text{want}}$

Correspondence at input nodes holds for equal indices

$$\Rightarrow R^{\text{got}} = R^{\text{want}} \cap \{i \leftrightarrow i\}$$

$$\Rightarrow R^{\text{lost}} = R^{\text{want}} \setminus \{i \leftrightarrow i\}$$

Backpropagation

$$\bullet R_{u_1, u_2}^{\text{lost}} = \bigcup_{(u_1, v_1), (u_2, v_2) \text{ s.t. } p_{(u_1, v_1)} = p_{(u_2, v_2)}} \left(M_{(u_1, v_1)}^{-1} \oplus M_{(u_2, v_2)}^{-1} \right) R_{v_1, v_2}^{\text{lost}}$$

$$\bullet R^{\text{got}} = R^{\text{want}} \setminus R^{\text{lost}}$$

Propagation of Correspondences

```
for (i = 0; i <= N; ++i)
    a[i] = in[i];
a[0] = in[1];
for (i = 0; i <= N; ++i)
    b[i] = f(a[N-i]);
for (i = 0; i <= N; ++i)
    Out(b[i]);
```

```
for (i = 0; i <= N; ++i) {
    a[i] = in[i];
    b[i] = f(a[i])
}
for (i = 0; i <= N; ++i)
    Out(b[N-i]);
```

Equivalence Checking Example

```
for (i = 0; i < M; ++i) {
    m = i+1;
    for (j = 0; j < N; ++j)
        m = g(h(m), in1[i][j]);
    compute_row(h(m), A[M-i-1]);
}
A[5][6] = 0;
for (i = 0; i < M - 2; ++i)
    out[i] = f(A[M-i-2-in2[i]]);

for (i = 0; i < M; ++i) {
    m = h(i+1);
    for (j = 0; j < N; ++j)
        m = h(g(m, in1[i][j]));
    compute_row(m, B[i]);
    if (i >= 2)
        out[i-2]=f(B[i-1+in2[i-2]]);
}
```

Can the basic approach prove equivalence of the two programs on the left?

Equivalence Checking Example

```
for (i = 0; i < M; ++i) {
    m = i+1;
    for (j = 0; j < N; ++j)
        m = g(h(m), in1[i][j]);
    compute_row(h(m), A[M-i-1]);
}
A[5][6] = 0;
for (i = 0; i < M - 2; ++i)
    out[i] = f(A[M-i-2-in2[i]]);

for (i = 0; i < M; ++i) {
    m = h(i+1);
    for (j = 0; j < N; ++j)
        m = h(g(m, in1[i][j]));
    compute_row(m, B[i]);
    if (i >= 2)
        out[i-2]=f(B[i-1+in2[i-2]]);
}
```

Can the basic approach prove equivalence of the two programs on the left?

⇒ No

Basic approach assumes

- data-independent index expressions
⇒ no data-dependent reads
- accesses to individual elements
⇒ no row accesses

Equivalence Checking Example

```
for (i = 0; i < M; ++i) {
    m = i+1;
    for (j = 0; j < N; ++j)
        m = g(h(m), in1[i][j]);
    compute_row(h(m), A[M-i-1]);
}
A[5][6] = 0;
for (i = 0; i < M - 2; ++i)
    out[i] = f(A[M-i-2-in2[i]]);

for (i = 0; i < M; ++i) {
    m = h(i+1);
    for (j = 0; j < N; ++j)
        m = h(g(m, in1[i][j]));
    compute_row(m, B[i]);
    if (i >= 2)
        out[i-2] = f(B[i-1+in2[i-2]]);
}
```

Can the basic approach prove equivalence of the two programs on the left?

⇒ No

Basic approach assumes

- data-independent index expressions
⇒ no data-dependent reads
- accesses to individual elements
⇒ no row accesses

Equivalence Checking Example

```
for (i = 0; i < M; ++i) {
    m = i+1;
    for (j = 0; j < N; ++j)
        m = g(h(m), in1[i][j]);
    compute_row(h(m), A[M-i-1]);
}
A[5][6] = 0;
for (i = 0; i < M - 2; ++i)
    out[i] = f(A[M-i-2-in2[i]]);

for (i = 0; i < M; ++i) {
    m = h(i+1);
    for (j = 0; j < N; ++j)
        m = h(g(m, in1[i][j]));
    compute_row(m, B[i]);
    if (i >= 2)
        out[i-2] = f(B[i-1+in2[i-2]]);
}
```

Can the basic approach prove equivalence of the two programs on the left?

⇒ No

Basic approach assumes

- data-independent index expressions
⇒ no data-dependent reads
- accesses to individual elements
⇒ no row accesses

Motivation for Extensions

Basic approach is unable to handle realistic multimedia systems

Multimedia code often contains

- Row accesses

For example, reads and writes to rows or even entire arrays in MP3:

```
for (j = 0; j <= 31; j++)  
    MPG_IMDCT_Win(buffer[j], rawout);
```

- Data-dependent reads

For example, data-dependent addressing in motion estimation of QSDPCM:

```
dist = f(..., prev_sub2_frame [...  
    +176*v4x[11*j+1]+2*v4y[11*j+1]]);
```

Expansion Edges

Dependence relations in basic approach map given use to **single** definition
Fails in presence of row reads:

```
for (i = 0; i < M; ++i) {  
    for (j = 0; j < N; ++j)  
        A[j] = g(/*...*/);  
    B[i] = f(A);  
}
```

Every iteration of f depends on N iterations of g
 $\Rightarrow$ introduction of *expansion* edges

Expansion Edges

Dependence relations in basic approach map given use to **single** definition

Fails in presence of row reads:

```
for (i = 0; i < M; ++i) {  
  for (j = 0; j < N; ++j)  
    A[j] = g(/*...*/);  
  B[i] = f(A);  
}
```

Every iteration of f depends on N iterations of g

$\Rightarrow$ introduction of *expansion edges*

Expansion Edges

Dependence relations in basic approach map given use to **single** definition
Fails in presence of row reads:

```
for (i = 0; i < M; ++i) {  
    for (j = 0; j < N; ++j)  
        A[j] = g(/*...*/);  
    B[i] = f(A);  
}
```

Every iteration of f depends on N iterations of g

$\Rightarrow$ introduction of *expansion edges*

Propagation over expansion edges requires special care

- either *neither* of the edges is an expansion or *both* are
- expansions should be the *same* (identical extra dimensions)
- only *corresponding* elements should be required to be equivalent

Propagation of $\{i \leftrightarrow i'\}$ over two expansion edges results in
 $\{(i, j) \leftrightarrow (i', j') \mid 0 \leq j, j' < N \wedge j = j'\}$

Data-dependent Reads

```
for (i = 0; i < M - 2; ++i)
    out[i] = f(A[M-i-2-in2[i]]);
```

Assume bounds are known on values of nested constructs, e.g., values of `in2[i]` range between -1 and 1 .

Value of data-dependent read is certainly the same if

- nested constructs have the same values, and
- same values are read for all values of nested constructs

Data-dependent Reads

```
for (i = 0; i < M - 2; ++i)
    out[i] = f(A[M-i-2-in2[i]]);
```

Assume bounds are known on values of nested constructs,
e.g., values of `in2[i]` range between -1 and 1 .

Value of data-dependent read is certainly the same if

- nested constructs have the same values, and
- same values are read for all values of nested constructs

⇒ introduce extra computation with $n + 1$ children

- n computations for each of the n nested constructs
- 1 computation with expanded domain ranging over all possible values

Data-dependent Reads

```
for (i = 0; i < M - 2; ++i)
    out[i] = f(A[M-i-2-in2[i]]);
```

Assume bounds are known on values of nested constructs, e.g., values of $\text{in2}[i]$ range between -1 and 1 .

Value of data-dependent read is certainly the same if

- nested constructs have the same values, and
- same values are read for all values of nested constructs

⇒ introduce extra computation with $n + 1$ children

- n computations for each of the n nested constructs
- 1 computation with expanded domain ranging over all possible values

Data-dependent Reads

```
for (i = 0; i < M - 2; ++i)
    out[i] = f(A[M-i-2-in2[i]]);
```

Assume bounds are known on values of nested constructs, e.g., values of $\text{in2}[i]$ range between -1 and 1 .

Value of data-dependent read is certainly the same if

- nested constructs have the same values, and
- same values are read for all values of nested constructs

⇒ introduce extra computation with $n + 1$ children

- n computations for each of the n nested constructs
- 1 computation with expanded domain ranging over all possible values

Data-dependent Reads

```
for (i = 0; i < M - 2; ++i)
    out[i] = f(A[M-i-2-in2[i]]);
```

Assume bounds are known on values of nested constructs, e.g., values of $\text{in2}[i]$ range between -1 and 1 .

Value of data-dependent read is certainly the same if

- nested constructs have the same values, and
- **same values are read for all values of nested constructs**

⇒ introduce extra computation with $n + 1$ children

- n computations for each of the n nested constructs
- 1 computation with expanded domain ranging over all possible values

Row Reads

```
for (i = 0; i < M - 2; ++i)
    out[i] = f(A[M-i-2-in2[i]]);
```

(A: array with rows of size N)

Value of row the same if

- values of all elements in the row are the same

Row Reads

```
for (i = 0; i < M - 2; ++i)
    out[i] = f(A[M-i-2-in2[i]]);
```

(A: array with rows of size N)

Value of row the same if

- values of all elements in the row are the same

⇒ introduce extra computation copying individual array elements
(edges leaving computation determined by dataflow analysis on elements)

Row Reads

```
for (i = 0; i < M - 2; ++i)
    out[i] = f(A[M-i-2-in2[i]]);
```

(A: array with rows of size N)

Value of row the same if

- values of all elements in the row are the same

⇒ introduce extra computation copying individual array elements
(edges leaving computation determined by dataflow analysis on elements)

Row Reads

```
for (i = 0; i < M - 2; ++i)
    out[i] = f(A[M-i-2-in2[i]]);
```

(A: array with rows of size N)

Value of row the same if

- values of all elements in the row are the same

⇒ introduce extra computation copying individual array elements
(edges leaving computation determined by dataflow analysis on elements)

Row Writes

```
for (i = 0; i < M; ++i) {  
    /* ... */  
    compute_row(h(m), A[M-i-1]);  
}
```

Dataflow analysis assumes writes to individual array elements

Row Writes

```
for (i = 0; i < M; ++i) {  
    /* ... */  
    compute_row(h(m), A[M-i-1]);  
}
```

Dataflow analysis assumes writes to individual array elements

⇒ introduce extra computation extracting elements from row

⇒ same row value

⇒ same index in row

Row Writes

```
for (i = 0; i < M; ++i) {  
    /* ... */  
    compute_row(h(m), A[M-i-1]);  
}
```

Dataflow analysis assumes writes to individual array elements
⇒ introduce extra computation extracting elements from row

⇒ same row value

⇒ same index in row

$a : \text{compute_row}/2$

$e : \text{id}/1$

Row Writes

```
for (i = 0; i < M; ++i) {  
    /* ... */  
    compute_row(h(m), A[M-i-1]);  
}
```

Dataflow analysis assumes writes to individual array elements
⇒ introduce extra computation extracting elements from row

⇒ same row value

⇒ same index in row

Row Writes

```
for (i = 0; i < M; ++i) {  
    /* ... */  
    compute_row(h(m), A[M-i-1]);  
}
```

Dataflow analysis assumes writes to individual array elements
⇒ introduce extra computation extracting elements from row

⇒ same row value

⇒ same index in row

Row Writes

```
for (i = 0; i < M; ++i) {  
    /* ... */  
    compute_row(h(m), A[M-i-1]);  
}
```

Dataflow analysis assumes writes to individual array elements
⇒ introduce extra computation extracting elements from row

⇒ same row value

⇒ same index in row

Implementation Details

All operations on sets/relations of integers performed by `isl`

- intersection
- union
- application of a map (relation) to a set, the domain of a relation or the range of a relation
- inverse of a relation (or application of an inverse map)
- integer affine hull (widening)
- subset testing
- equality testing
- emptiness testing

Implementation Details

All operations on sets/relations of integers performed by `isl`

- intersection
- union
- application of a map (relation) to a set, the domain of a relation or the range of a relation
- inverse of a relation (or application of an inverse map)
- integer affine hull (widening)
- subset testing
- equality testing
- emptiness testing

Other features

- explicit support for existentially quantified variables
- all computations in exact integer arithmetic using GMP
- built-in incremental LP solver
- built-in parametric integer programming solver
- released under LGPL license

Experimental Results

Program pair	equiv.	comps	d	row	dd	ec time	∇	Δ
CLooG test 1	✓	135	3			0.222s	51	0
CLooG test 2	✓	1092	3			5.574s	116	0
CLooG test 3	✓	28	5			0.487s	29	0
CLooG test 4	✓	50	2			16.772s	201	0
CLooG test suite	✓	4618	5			37.589s	1747	0
Motivating ex.		26	3	✓	✓	0.036s	1	0
MP3	✓	58	4	✓		0.037s	2	0
MP3'		59	4	✓		0.072s	2	14
QSDPCM	✓	252	8		✓	40.270s	259	0

(isa-0.09-35-g24a7f0c on Intel Xeon W3520 @ 2.66GHz)

Conclusions

- Basic approach
 - ▶ uses widening instead of transitive closure for handling recurrences
 - ▶ supports non-uniform recurrences
 - ▶ supports associative and commutative operations with fixed number of arguments
- Extensions
 - ▶ expansion edges
 - ▶ data-dependent read accesses
 - ▶ row read and row write accesses
- Publicly available prototype implementation
(<http://www.kotnet.org/~skimo/loop/isa-0.09.tar.bz2>)

Future work

- data-dependent writes and data-dependent assignments
- associative operations with a variable number of arguments, e.g., different encodings of $\sum_{0 \leq i < N} a[i]$
- general reductions
- data-dependent and non-affine conditions and index expressions