On Demand Parametric Array Dataflow Analysis

Sven Verdoolaege Hristo Nikolov Todor Stefanov

Leiden Institute for Advanced Computer Science
École Normale Supérieure and INRIA

January 21, 2013

Outline

Outline

Motivation

- Dataflow analysis determines for read access in a statement instance, the statement instance that wrote the value being read
- Many uses in polyhedral analysis/compilation
  - array expansion
  - scheduling
  - equivalence checking
  - optimizing computation/communication overlap in MPI programs
  - derivation of process networks
  - ...
- Standard dataflow analysis (Feautrier) requires static affine input programs
- Extensions are needed for programs with dynamic/non-affine constructs

Motivation

- Dataflow analysis determines for read access in a statement instance, the statement instance that wrote the value being read
- Many uses in polyhedral analysis/compilation
  - array expansion
  - scheduling
  - equivalence checking
  - optimizing computation/communication overlap in MPI programs
  - derivation of process networks
  - ...
- Standard dataflow analysis (Feautrier) requires static affine input programs
- Extensions are needed for programs with dynamic/non-affine constructs

Our Motivation: Derivation of Process Networks

- Main purpose: extract task level parallelism from dataflow graph

$$\begin{array}{rcl} \text{statement} & \rightarrow & \text{process} \\ \text{flow dependence} & \rightarrow & \text{communication channel} \end{array}$$

  $\Rightarrow$ requires dataflow analysis
- Processes are mapped to parallel hardware (e.g., FPGA)

Our Motivation: Derivation of Process Networks

- Main purpose: extract task level parallelism from dataflow graph

$$\begin{array}{rcl} \text{statement} & \rightarrow & \text{process} \\ \text{flow dependence} & \rightarrow & \text{communication channel} \end{array}$$

  $\Rightarrow$ requires dataflow analysis

- Processes are mapped to parallel hardware (e.g., FPGA)

Example:

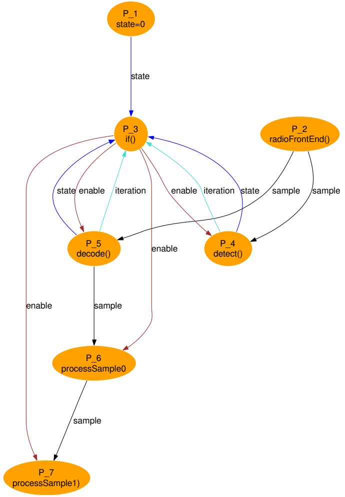
```
for (i = 0; i < n; ++i) {
        a = f();
        g(a);
}
```

Dynamic Process Networks

```
int state = 0;
for (i = 0; i <= 10; i++) {
  sample = radioFrontend();
  if (state == 0) {
    state = detect(sample);
  } else {
    state = decode(sample, &value0);
    value1 = processSample0(value0);
    processSample1(value1);
  }
}
```

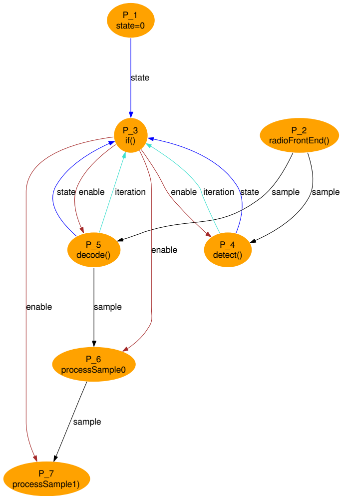
Dynamic Process Networks

```
int state = 0;
for (i = 0; i <= 10; i++) {
  sample = radioFrontend();
  if (state == 0) {
    state = detect(sample);
  } else {
    state = decode(sample, &value0);
    value1 = processSample0(value0);
    processSample1(value1);
  }
}
```



Dynamic Process Networks

```
int state = 0;
for (i = 0; i <= 10; i++) {
  sample = radioFrontend();
  if (state == 0) {
    state = detect(sample);
  } else {
    state = decode(sample, &value0);
    value1 = processSample0(value0);
    processSample1(value1);
  }
}
```



- additional control channels
- determine operation of data channels
- dataflow analysis needs to remain exact, but may depend on run-time information

Outline

Standard Array Dataflow Analysis

Given a read from an array element, what was the last write to the same array element before the read?

Simple case: array written through a single reference

```
for (i = 0; i < N; ++i)
    for (j = 0; j < N - i; ++j)
F:      a[i+j] = f(a[i+j]);
for (i = 0; i < N; ++i)
W:  Write(a[i]);
```

Standard Array Dataflow Analysis

Given a read from an array element, what was the last write to the same array element before the read?

Simple case: array written through a single reference

```
for (i = 0; i < N; ++i)
    for (j = 0; j < N - i; ++j)
F:      a[i+j] = f(a[i+j]);
for (i = 0; i < N; ++i)
W:  Write(a[i]);
```

Standard Array Dataflow Analysis

Given a read from an array element, what was the last write to the same array element before the read?

Simple case: array written through a single reference

```
for (i = 0; i < N; ++i)
    for (j = 0; j < N - i; ++j)
F:      a[i+j] = f(a[i+j]);
for (i = 0; i < N; ++i)
W:  Write(a[i]);
```

Access relations:

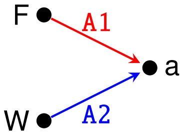
```
A1:=[N]->{F[i,j]->a[i+j]:0<=i<N and 0<=j<N-i};
A2:=[N]->{W[i] -> a[i] : 0 <= i < N };
```

Standard Array Dataflow Analysis

Given a read from an array element, what was the last write to the same array element before the read?

Simple case: array written through a single reference

```
for (i = 0; i < N; ++i)
    for (j = 0; j < N - i; ++j)
F:      a[i+j] = f(a[i+j]);
for (i = 0; i < N; ++i)
W:  Write(a[i]);
```



Access relations:

```
A1:=[N]->{F[i,j]->a[i+j]:0<=i<N and 0<=j<N-i};
A2:=[N]->{W[i] -> a[i] : 0 <= i < N };
```

Map to all writes: `R := A2 . (A1^-1);`

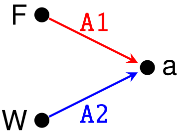
```
[N] -> { W[i] -> F[i',i-i'] : 0 <= i,i'< N and i'<= i }
```

Standard Array Dataflow Analysis

Given a read from an array element, what was the last write to the same array element before the read?

Simple case: array written through a single reference

```
for (i = 0; i < N; ++i)
    for (j = 0; j < N - i; ++j)
F:      a[i+j] = f(a[i+j]);
for (i = 0; i < N; ++i)
W:  Write(a[i]);
```



Access relations:

```
A1:=[N]->{F[i,j]->a[i+j]:0<=i<N and 0<=j<N-i};
A2:=[N]->{W[i] -> a[i] : 0 <= i < N };
```

Map to all writes: `R := A2 . (A1^-1);`

```
[N] -> { W[i] -> F[i',i-i'] : 0 <= i,i'< N and i'<= i }
```

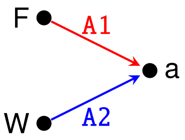
Last write: `lexmax R; # [N] -> { W[i] -> F[i,0] : 0 <= i < N }`

Standard Array Dataflow Analysis

Given a read from an array element, what was the last write to the same array element before the read?

Simple case: array written through a single reference

```
for (i = 0; i < N; ++i)
    for (j = 0; j < N - i; ++j)
F:          a[i+j] = f(a[i+j]);
for (i = 0; i < N; ++i)
W:  Write(a[i]);
```



Access relations:

```
A1:=[N]->{F[i,j]->a[i+j]:0<=i<N and 0<=j<N-i};
A2:=[N]->{W[i] -> a[i] : 0 <= i < N };
```

Map to all writes: `R := A2 . (A1^-1);`

`[N] -> { W[i] -> F[i',i-i'] : 0 <= i,i'< N and i'<= i }`

Last write: `lexmax R; # [N] -> { W[i] -> F[i,0] : 0 <= i < N }`

In general: impose lexicographical order on shared iterators

Standard Array Dataflow Analysis

Multiple Potential Sources

- Dataflow is typically performed per read access ("sink") C
- Corresponding writes ("potential sources") P are considered in turn
- Map to all potential source iterations: $D^{\mathrm{mem}}_{C,P} = (A_P^{-1} \circ A_C) \cap B_C^P$ ("memory based dependences"; B_C^P: P executed before C)
- Source may already be known for some sink iterations
  $\Rightarrow$ compute *partial* lexicographical maximum

$$(U', D) = \underset{U}{\operatorname{lexmax}} M$$

U: sink iterations for which no source has been found
M: part of memory based dependences for particular potential source
$U' = U \setminus \operatorname{dom} M$
$M' = \operatorname{lexmax}(M \cap (U \to \operatorname{ran} M))$

Note: here, dependence relations map sink iterations to source iterations

Fuzzy Array Dataflow Analysis

- Introduces parameters for each lexmax involving dynamic behavior
- Parameters represent dynamic solution of lexmax operation
- Derives properties on parameters after dataflow analysis (using resolution)

Fuzzy Array Dataflow Analysis

- Introduces parameters for each lexmax involving dynamic behavior
- Parameters represent dynamic solution of lexmax operation
- Derives properties on parameters after dataflow analysis (using resolution)

- Parametric result is exact
- Parameters can be projected out to obtain approximate but static dataflow

Fuzzy Array Dataflow Analysis

- Introduces parameters for each lexmax involving dynamic behavior
- Parameters represent dynamic solution of lexmax operation
- Derives properties on parameters after dataflow analysis (using resolution)

- Parametric result is exact
- Parameters can be projected out to obtain approximate but static dataflow

Main problem for deriving process networks:
Introduces too many parameters
$\Rightarrow$ too many control channels

On Demand Parametric Array Dataflow Analysis

Similar to FADA:

- Exact, possibly parametric, dataflow
- Introduces parameters to represent dynamic behavior

But:

+ Parameters have a different meaning
+ Effect analyzed before parameters are introduced
+ All computations are performed directly on affine sets and maps
– Currently only supports dynamic conditions

Outline

Representing Generic Dynamic Conditions

```
   while (1) {
       sample = radioFrontend();
       if (t(state)) {
D:         state = detect(sample);
       } else { /* ... */ }
   }
```

Representing Generic Dynamic Conditions

```
  while (1) {
      sample = radioFrontend();
      if (t(state)) {
D:          state = detect(sample);
      } else { /* ... */ }
  }
```

Dynamic condition (`t(state)`) represented by *filter*

- Filter access relation(s):
  access to (virtual) array representing condition

$$\{\, D(i) \to (S_0(i) \to t_0(i)) \,\}$$

- Filter value relation:
  values of filter array elements for which statement is executed

$$\{\, \mathtt{D}(i) \to (1) \mid i \geq 0 \,\}$$

Representing Generic Dynamic Conditions

```
   while (1) {
       sample = radioFrontend();
       if (t(state)) {          S0: t0 = t(state);
D:         state = detect(sample);
       } else { /* ... */ }
   }
```

Dynamic condition (`t(state)`) represented by *filter*

- Filter access relation(s):
  access to (virtual) array representing condition

$$\{ D(i) \to (S_0(i) \to t_0(i)) \}$$

- Filter value relation:
  values of filter array elements for which statement is executed

$$\{ \mathtt{D}(i) \to (1) \mid i \geq 0 \}$$

Representing Generic Dynamic Conditions

```
   while (1) {
       sample = radioFrontend();
       if (t(state)) {                  S0: t0 = t(state);
D:         state = detect(sample);
       } else { /* ... */ }
   }
```

Dynamic condition (`t(state)`) represented by *filter*

- Filter access relation(s):
  access to (virtual) array representing condition

filter array

$$\{ D(i) \to (S_0(i) \to \boxed{t_0}(i)) \}$$

- Filter value relation:
  values of filter array elements for which statement is executed

$$\{ \mathrm{D}(i) \to (1) \mid i \geq 0 \}$$

Representing Generic Dynamic Conditions

```
   while (1) {
       sample = radioFrontend();
       if (t(state)) {                  S0: t0 = t(state);
D:         state = detect(sample);
       } else { /* ... */ }
   }
```

Dynamic condition (`t(state)`) represented by *filter*

- Filter access relation(s): statement writing to filter array
  access to (virtual) array representing condition filter array

$$\{\, D(i) \to (\boxed{S_0}(i) \to \boxed{t_0}(i)) \,\}$$

- Filter value relation:
  values of filter array elements for which statement is executed

$$\{\, \mathrm{D}(i) \to (1) \mid i \geq 0 \,\}$$

Representing Generic Dynamic Conditions

```
   while (1) {
        sample = radioFrontend();
        if (t(state)) {          S0: t0 = t(state);
D:           state = detect(sample);
        } else { /* ... */ }
   }
```

Dynamic condition (`t(state)`) represented by *filter*

- Filter access relation(s): statement writing to filter array
  access to (virtual) array representing condition filter array

$$\{\, \boxed{D}(i) \to (\boxed{S_0}(i) \to \boxed{t_0}(i))\,\}$$

statement reading from filter array

- Filter value relation:
  values of filter array elements for which statement is executed

$$\{\, \mathrm{D}(i) \to (1) \mid i \geq 0 \,\}$$

Representing Locally Static Affine Conditions

```
N1: n = f();
    for (int k = 0; k < 100; ++k) {
M:      m = g();
        for (int i = 0; i < m; ++i)
            for (int j = 0; j < n; ++j)
A:              a[j][i] = g();
N2:     n = f();
    }
```

Values of `m` and `n` not changed inside `i` and `j` loops

$\Rightarrow$ locally static affine loop conditions

Representing Locally Static Affine Conditions

```
N1: n = f();
    for (int k = 0; k < 100; ++k) {
M:      m = g();
        for (int i = 0; i < m; ++i)
            for (int j = 0; j < n; ++j)
A:              a[j][i] = g();
N2:     n = f();
    }
```

Values of `m` and `n` not changed inside `i` and `j` loops
$\Rightarrow$ locally static affine loop conditions

- Filter access relations:
  $\{\, \mathtt{A}(k,i,j) \to (\mathtt{M}(k) \to \mathtt{m}()) \,\}$
  $\{\, \mathtt{A}(0,i,j) \to (\mathtt{N1}() \to \mathtt{n}()) \,\} \cup \{\, \mathtt{A}(k,i,j) \to (\mathtt{N2}(k-1) \to \mathtt{n}()) \mid k \geq 1 \,\}$
- Filter value relation:
  $\{\, \mathtt{A}(k,i,j) \to (m,n) \mid 0 \leq k \leq 99 \wedge 0 \leq i < m \wedge 0 \leq j < n \,\}$

Note: filter access relations exploit (static) dataflow analysis on `m` and `n`

Outline

Overview

- Dataflow analysis performed for each read access (sink) separately
- Potential sources considered from closest to furthest
  - number of shared loop iterators ℓ
  - textual order
- For each lexmax operation
  - is it possible for potential source not to execute when sink is executed? (based on filters)
  - if so, parametrize lexmax problem

Parametrization

```
   state = 0;
   while (1) {
       sample = radioFrontend();
       if (t(state)) {
D:         state = detect(sample);
       } else {
C:         decode(sample, &state, &value0);
           value1 = processSample0(value0);
           processSample1(value1);
       }
   }
```

Parametrization

```
   state = 0;
   while (1) {
        sample = radioFrontend();
        if (t(state)) {
D:          state = detect(sample);
        } else {
C:          decode(sample, &state, &value0);
            value1 = processSample0(value0);
            processSample1(value1);
        }
   }
```

sink *C*

potential source *P*

Parametrization

```
   state = 0;
   while (1) {
        sample = radioFrontend();
        if (t(state)) {
D:           state = detect(sample);
        } else {
C:           decode(sample, &state, &value0);
             value1 = processSample0(value0);
             processSample1(value1);
        }
   }
```

sink C

potential source P

Memory based dependences: $D^{\text{mem}}_{C,P} = \{\, \mathtt{S_0}(i) \to \mathtt{D}(i') \mid 0 \le i' < i \,\}$

At $\ell = 1$: $M = D^{\text{mem}}_{C,P} \cap \{\, \mathtt{S_0}(i) \to \mathtt{D}(i) \,\} = \emptyset$

At $\ell = 0$: $M = \{\, \mathtt{S_0}(i) \to \mathtt{D}(i') \mid 0 \le i' < i \,\}$

Potential source $\mathtt{D}(i')$ may not have executed even if sink $\mathtt{S_0}(i)$ is executed

$\Rightarrow$ parametrization required

Parameter Representation

Original:

$$M = \{\, \mathtt{S_0}(i) \to \mathtt{D}(i') \mid 0 \le i' < i \,\}$$

After parameter introduction:

$$M' = \left\{ \mathtt{S_0}(i) \to \mathtt{D}(\lambda_C^P(i)) \mid 0 \le \lambda_C^P(i) < i \wedge \beta_C^P(i) = 1 \right\}$$

$\Rightarrow \operatorname{lexmax} M' = M'$

Parameter Representation

Original:

$$M = \{\, \mathtt{S}_0(i) \to \mathtt{D}(i') \mid 0 \le i' < i \,\}$$

After parameter introduction:

$$M' = \left\{ \mathtt{S}_0(i) \to \mathtt{D}(\lambda_C^P(i)) \mid 0 \le \lambda_C^P(i) < i \wedge \beta_C^P(i) = 1 \right\}$$

$\Rightarrow \operatorname{lexmax} M' = M'$

Meaning of the parameters:

- $\lambda_C^P(\mathbf{k})$: last executed iteration of $D_{C,P}^{\mathrm{mem}}(\mathbf{k})$
- $\beta_C^P(\mathbf{k})$: any iteration of $D_{C,P}^{\mathrm{mem}}(\mathbf{k})$ is executed

Note: FADA introduces separate set of parameters for each lexmax
Note: $\lambda_C^P(\mathbf{k})$ and $\beta_C^P(\mathbf{k})$ depend on $\mathbf{k}$, but dependence can be kept implicit
$\Rightarrow \lambda_C^P$ and β_C^P

Introducing as few Parameters as possible

In principle, the number of elements in λ is equal to the number of iterators
However, in many cases, we can avoid introducing some of those elements

Introducing as few Parameters as possible

In principle, the number of elements in λ is equal to the number of iterators
However, in many cases, we can avoid introducing some of those elements

- dimensions inside innermost condition that is not static affine

Introducing as few Parameters as possible

In principle, the number of elements in λ is equal to the number of iterators
However, in many cases, we can avoid introducing some of those elements

- dimensions inside innermost condition that is not static affine

```
for (i = 0; i < 100; ++i)
        if (t())
                for (j = 0; j < 100; ++j)
A:                      a = t();
B: b = a;
```

$$M = \{ B() \to A(i,j) \mid 0 \le i,j < 100 \}$$

Introducing as few Parameters as possible

In principle, the number of elements in λ is equal to the number of iterators
However, in many cases, we can avoid introducing some of those elements

- dimensions inside innermost condition that is not static affine

```
for (i = 0; i < 100; ++i)
        if (t())
                for (j = 0; j < 100; ++j)
A:                      a = t();
B: b = a;
```

$$M = \{ B() \to A(i,j) \mid 0 \le i,j < 100 \}$$

$$M' = \{ B() \to A(\lambda_0,j) \mid 0 \le \lambda_0, j < 100 \wedge \beta = 1 \}$$

Introducing as few Parameters as possible

In principle, the number of elements in λ is equal to the number of iterators
However, in many cases, we can avoid introducing some of those elements

- dimensions inside innermost condition that is not static affine

```
for (i = 0; i < 100; ++i)
        if (t())
                for (j = 0; j < 100; ++j)
A:                      a = t();
B: b = a;
```

$$M = \{ B() \to A(i,j) \mid 0 \le i,j < 100 \}$$

$$M' = \{ B() \to A(\lambda_0,j) \mid 0 \le \lambda_0, j < 100 \wedge \beta = 1 \}$$

$$\mathrm{lexmax}\, M' = \{ B() \to A(\lambda_0, 99) \mid 0 \le \lambda_0 < 100 \wedge \beta = 1 \}$$

Introducing as few Parameters as possible

In principle, the number of elements in λ is equal to the number of iterators
However, in many cases, we can avoid introducing some of those elements

- dimensions inside innermost condition that is not static affine
- dimensions that can only attain a single value (for a given value of $\mathbf{k}$)

Introducing as few Parameters as possible

Dimensions that can only attain a single value

```
for (int k = 0; k < 100; ++k) {
N:        N = f();
M:        M = g();
          for (int i = 0; i < N; ++i)
                    for (int j = 0; j < M; ++j)
A:                            a[i][j] = i + j;
          for (int i = 0; i < N; ++i)
                    for (int j = 0; j < M; ++j)
H:                            h(i, j, a[i][j]);
}
```

$$D^{\mathrm{mem}}_{\mathrm{H,A}} = \{\, \mathtt{H}(k,i,j) \to \mathtt{A}(k',i,j) \mid k' \le k \,\}$$

$$\lambda_1(k,i,j) = i$$

$$\lambda_2(k,i,j) = j$$

$\Rightarrow$ no need to introduce λ_1 and λ_2

Introducing as few Parameters as possible

In principle, the number of elements in λ is equal to the number of iterators
However, in many cases, we can avoid introducing some of those elements

- dimensions inside innermost condition that is not static affine
- dimensions that can only attain a single value (for a given value of **k**)

Introducing as few Parameters as possible

In principle, the number of elements in λ is equal to the number of iterators
However, in many cases, we can avoid introducing some of those elements

- dimensions inside innermost condition that is not static affine
- dimensions that can only attain a single value (for a given value of $\mathbf{k}$)
- dimensions before ℓ

Introducing as few Parameters as possible

Dimensions before ℓ

```
for (int k = 0; k < 100; ++k) {
N:        N = f();
M:        M = g();
          for (int i = 0; i < N; ++i)
                  for (int j = 0; j < M; ++j)
A:                        a[i][j] = i + j;
          for (int i = 0; i < N; ++i)
                  for (int j = 0; j < M; ++j)
H:                        h(i, j, a[i][j]);
}
```

At $\ell = 1$:

$$M = \{\, \mathtt{H}(k, i, j) \to \mathtt{A}(k, i, j) \,\}$$

$\Rightarrow$ no need to introduce λ_0 (yet) at $\ell = 1$

Introducing as few Parameters as possible

Dimensions before ℓ

```
for (int k = 0; k < 100; ++k) {
N:        N = f();
M:        M = g();
          for (int i = 0; i < N; ++i)
                    for (int j = 0; j < M; ++j)
A:                            a[i][j] = i + j;
          for (int i = 0; i < N; ++i)
                    for (int j = 0; j < M; ++j)
H:                            h(i, j, a[i][j]);
}
```

At $\ell = 1$:

$$M = \{ \mathtt{H}(k, i, j) \to \mathtt{A}(k, i, j) \}$$

$\Rightarrow$ no need to introduce λ_0 (yet) at $\ell = 1$

Note: all sinks are accounted for at $\ell = 1$

$\Rightarrow$ no need to consider $\ell = 0$ and λ_0 not needed at all

Introducing as few Parameters as possible

In principle, the number of elements in λ is equal to the number of iterators
However, in many cases, we can avoid introducing some of those elements

- dimensions inside innermost condition that is not static affine
- dimensions that can only attain a single value (for a given value of **k**)
- dimensions before ℓ

Introducing as few Parameters as possible

In principle, the number of elements in λ is equal to the number of iterators
However, in many cases, we can avoid introducing some of those elements

- dimensions inside innermost condition that is not static affine
- dimensions that can only attain a single value (for a given value of **k**)
- dimensions before ℓ

  $\Rightarrow$ replace β by σ: the number of implicitly equal shared iterators

$$
\begin{aligned}
\beta = 1 \quad &\rightarrow \quad \sigma \geq \ell \\
\beta = 0 \quad &\rightarrow \quad \sigma < \ell
\end{aligned}
$$

Introducing as few Parameters as possible

In principle, the number of elements in λ is equal to the number of iterators
However, in many cases, we can avoid introducing some of those elements

- dimensions inside innermost condition that is not static affine
- dimensions that can only attain a single value (for a given value of $\mathbf{k}$)
- dimensions before ℓ

  $\Rightarrow$ replace β by σ: the number of implicitly equal shared iterators

$$\begin{array}{lcl} \beta = 1 & \rightarrow & \sigma \geq \ell \\ \beta = 0 & \rightarrow & \sigma < \ell \end{array}$$

  - when moving to $\ell - 1$
    - ★ introduce additional parameter $\lambda_{\ell-1}$ (if needed)
    - ★ make implicit equality explicit
  - at the end of the dataflow analysis

$$\begin{array}{lcl} \sigma \geq \ell_{\leq} & \rightarrow & \beta = 1 \\ \sigma < \ell_{\leq} & \rightarrow & \beta = 0 \end{array}$$

    ($\ell_{\leq}$: smallest ℓ for which parametrization was applied)
    $\lambda(\mathbf{k})$ and $\beta(\mathbf{k})$ now refer to last execution of $\overline{D}(\mathbf{k})$
    ($\overline{D}$: result of projecting out parameters from final dataflow relation)

When to Introduce Parameters

- Sink C
- Potential source P
- Subset of sink iteration U
- Mapping to potential source iterations M

Computing

$$(U', D) = \operatorname*{lexmax}_{U} M$$

When to Introduce Parameters

- Sink C
- Potential source P
- Subset of sink iteration U
- Mapping to potential source iterations M

Computing

$$(U', D) = \operatorname*{lexmax}_U M$$

1. No filter on source
   $\Rightarrow$ stop (no parametrization required)

When to Introduce Parameters

- Sink C
- Potential source P
- Subset of sink iteration U
- Mapping to potential source iterations M

1. No filter on source
   $\Rightarrow$ stop (no parametrization required)
2. Let F be the filter on the sink
3. Filter on source contradicts F
   $\Rightarrow$ replace M by empty relation and stop

Computing

$$(U', D) = \operatorname*{lexmax}_{U} M$$

Filter on source contradicts F

$\ell = 1$

```
while (1) {
N:   n = f();
     a = g();
     if (n < 100)
H:         a = h();
     if (n > 200)
T:         t(a);
}
```

potential source

sink

- Potential source filter access relation

$$\left\{ \mathtt{H}(i) \to (\mathtt{N}(i) \to \mathtt{n}) \right\}$$

- Potential source filter value relation

$$\left\{ \mathtt{H}(i) \to (n) \mid i \geq 0 \wedge n < 100 \right\}$$

- Sink filter access relation

$$\left\{ \mathtt{T}(i) \to (\mathtt{N}(i) \to \mathtt{n}) \right\}$$

- Sink filter value relation

$$\left\{ \mathtt{T}(i) \to (n) \mid i \geq 0 \wedge n > 200 \right\}$$

Filter on source contradicts F

$\ell = 1$

same filter element

```
while (1) {
N:   n = f();
     a = g();
     if (n < 100)
H:       a = h();
     if (n > 200)
T:       t(a);
}
```

potential source

sink

- Potential source filter access relation

$$\left\{ \mathtt{H}(i) \to (\boxed{\mathtt{N}(i) \to \mathtt{n}}) \right\}$$

- Potential source filter value relation

$$\left\{ \mathtt{H}(i) \to (n) \mid i \geq 0 \wedge n < 100 \right\}$$

- Sink filter access relation

$$\left\{ \mathtt{T}(i) \to (\boxed{\mathtt{N}(i) \to \mathtt{n}}) \right\}$$

- Sink filter value relation

$$\left\{ \mathtt{T}(i) \to (n) \mid i \geq 0 \wedge n > 200 \right\}$$

Filter on source contradicts F

$\ell = 1$

same filter element

```
while (1) {
N:   n = f();
     a = g();
     if (n < 100)
H:        a = h();
     if (n > 200)
T:        t(a);
}
```

potential source

sink

contradiction

- Potential source filter access relation

$$\left\{ \mathtt{H}(i) \to \left(\boxed{\mathtt{N}(i) \to \mathtt{n}} \right) \right\}$$

- Potential source filter value relation

$$\left\{ \mathtt{H}(i) \to (n) \mid i \geq 0 \wedge \boxed{n < 100} \right\}$$

- Sink filter access relation

$$\left\{ \mathtt{T}(i) \to \left(\boxed{\mathtt{N}(i) \to \mathtt{n}} \right) \right\}$$

- Sink filter value relation

$$\left\{ \mathtt{T}(i) \to (n) \mid i \geq 0 \wedge \boxed{n > 200} \right\}$$

When to Introduce Parameters

- Sink C
- Potential source P
- Subset of sink iteration U
- Mapping to potential source iterations M

1. No filter on source
   $\Rightarrow$ stop (no parametrization required)
2. Let F be the filter on the sink
3. Filter on source contradicts F
   $\Rightarrow$ replace M by empty relation and stop

Computing

$$(U', D) = \operatorname*{lexmax}_U M$$

When to Introduce Parameters

- Sink C
- Potential source P
- Subset of sink iteration U
- Mapping to potential source iterations M

Computing

$$(U', D) = \operatorname*{lexmax}_U M$$

1. No filter on source
   $\Rightarrow$ stop (no parametrization required)
2. Let F be the filter on the sink
3. Filter on source contradicts F
   $\Rightarrow$ replace M by empty relation and stop
4. Let F' be equal to F updated with information from other sources
5. Filter on source contradicts F'
   $\Rightarrow$ replace M by empty relation and stop

Filter on source contradicts F'

```
N:   n = f();
     if (n < 100)
H:       a = h();
     if (n < 200)
H2:      a = h2();
T:   t(a);
}
```

potential source

sink

$$\operatorname*{lexmax}_{U} M$$

$$M = \{\, \mathtt{T}() \to \mathtt{H}() \,\}$$

$$U = \left\{\, \mathtt{T}() \mid \sigma^{\mathrm{H2}} < 0 \,\right\}$$

Filter on source contradicts F'

```
N:   n = f();
     if (n < 100)
H:         a = h();
     if (n < 200)
H2:        a = h2();
T:   t(a);
}
```

potential source

sink

$$\operatorname*{lexmax}_{U} M$$

$$M = \{\,\mathtt{T}() \to \mathtt{H}()\,\}$$

$$U = \left\{\mathtt{T}() \mid \boxed{\sigma^{\mathtt{H2}} < 0}\right\}$$

H2 not executed

Filter on source contradicts F'

```
N:   n = f();
     if (n < 100)
H:        a = h();
     if (n < 200)
H2:       a = h2();
T:   t(a);
}
```

potential source

sink

$$\operatorname*{lexmax}_{U} M$$

$$M = \{\, \mathtt{T}() \to \mathtt{H}() \,\}$$

$$U = \left\{ \mathtt{T}() \mid \boxed{\sigma^{\mathtt{H2}} < 0} \right\}$$

H2 not executed

- Updated sink filter access relation

$$\left\{ \mathtt{T}(i) \to (\mathtt{N}(i) \to \mathtt{n}) \right\}$$

- Updated sink filter value relation

$$\left\{ \mathtt{T}(i) \to (n) \mid i \geq 0 \wedge n \geq 200 \right\}$$

When to Introduce Parameters

- Sink C
- Potential source P
- Subset of sink iteration U
- Mapping to potential source iterations M

Computing

$$(U', D) = \operatorname*{lexmax}_U M$$

1. No filter on source
   $\Rightarrow$ stop (no parametrization required)
2. Let F be the filter on the sink
3. Filter on source contradicts F
   $\Rightarrow$ replace M by empty relation and stop
4. Let F' be equal to F updated with information from other sources
5. Filter on source contradicts F'
   $\Rightarrow$ replace M by empty relation and stop

When to Introduce Parameters

- Sink C
- Potential source P
- Subset of sink iteration U
- Mapping to potential source iterations M

Computing

$$(U', D) = \operatorname*{lexmax}_{U} M$$

1. No filter on source
   $\Rightarrow$ stop (no parametrization required)
2. Let F be the filter on the sink
3. Filter on source contradicts F
   $\Rightarrow$ replace M by empty relation and stop
4. Let F' be equal to F updated with information from other sources
5. Filter on source contradicts F'
   $\Rightarrow$ replace M by empty relation and stop
6. Filter on source implied by F
   $\Rightarrow$ stop (no parametrization required)

Filter on source implied by F

$$\ell = 1$$

```
while (1) {
N:  n = f();
    a = g();
    if (n < 200)
H:       a = h();
    if (n < 100)
T:       t(a);
}
```

potential source

sink

- Potential source filter access relation

$$\left\{ \mathtt{H}(i) \to (\mathtt{N}(i) \to \mathtt{n}) \right\}$$

- Potential source filter value relation

$$\left\{ \mathtt{H}(i) \to (n) \mid i \geq 0 \wedge n < 200 \right\}$$

- Sink filter access relation

$$\left\{ \mathtt{T}(i) \to (\mathtt{N}(i) \to \mathtt{n}) \right\}$$

- Sink filter value relation

$$\left\{ \mathtt{T}(i) \to (n) \mid i \geq 0 \wedge n < 100 \right\}$$

When to Introduce Parameters

- Sink C
- Potential source P
- Subset of sink iteration U
- Mapping to potential source iterations M

Computing

$$(U', D) = \operatorname*{lexmax}_{U} M$$

1. No filter on source
   $\Rightarrow$ stop (no parametrization required)
2. Let F be the filter on the sink
3. Filter on source contradicts F
   $\Rightarrow$ replace M by empty relation and stop
4. Let F' be equal to F updated with information from other sources
5. Filter on source contradicts F'
   $\Rightarrow$ replace M by empty relation and stop
6. Filter on source implied by F
   $\Rightarrow$ stop (no parametrization required)

When to Introduce Parameters

- Sink C
- Potential source P
- Subset of sink iteration U
- Mapping to potential source iterations M

Computing

$$(U', D) = \operatorname*{lexmax}_{U} M$$

1. No filter on source
   $\Rightarrow$ stop (no parametrization required)
2. Let F be the filter on the sink
3. Filter on source contradicts F
   $\Rightarrow$ replace M by empty relation and stop
4. Let F' be equal to F updated with information from other sources
5. Filter on source contradicts F'
   $\Rightarrow$ replace M by empty relation and stop
6. Filter on source implied by F
   $\Rightarrow$ stop (no parametrization required)
7. Filter on source implied by F'
   $\Rightarrow$ parametrize D and stop

Filter on source implied by F'

```
N:   n = f();
     if (n < 200)
H:         a = h();
     if (n > 100)
H2:        a = h2();
T:   t(a);
}
```

potential source

sink

$$\operatorname*{lexmax}_{U} M$$

$$M = \{\, \mathtt{T}() \to \mathtt{H}() \,\}$$

$$U = \left\{\, \mathtt{T}() \mid \sigma^{\mathtt{H2}} < 0 \,\right\}$$

- Updated sink filter access relation

$$\left\{\, \mathtt{T}(i) \to (\mathtt{N}(i) \to \mathtt{n}) \,\right\}$$

- Updated sink filter value relation

$$\left\{\, \mathtt{T}(i) \to (n) \mid i \geq 0 \land n \leq 100 \,\right\}$$

When to Introduce Parameters

- Sink C
- Potential source P
- Subset of sink iteration U
- Mapping to potential source iterations M

Computing

$$(U', D) = \operatorname*{lexmax}_{U} M$$

1. No filter on source
   $\Rightarrow$ stop (no parametrization required)
2. Let F be the filter on the sink
3. Filter on source contradicts F
   $\Rightarrow$ replace M by empty relation and stop
4. Let F' be equal to F updated with information from other sources
5. Filter on source contradicts F'
   $\Rightarrow$ replace M by empty relation and stop
6. Filter on source implied by F
   $\Rightarrow$ stop (no parametrization required)
7. Filter on source implied by F'
   $\Rightarrow$ parametrize D and stop

When to Introduce Parameters

- Sink C
- Potential source P
- Subset of sink iteration U
- Mapping to potential source iterations M

Computing

$$(U', D) = \operatorname*{lexmax}_{U} M$$

1. No filter on source
   $\Rightarrow$ stop (no parametrization required)
2. Let F be the filter on the sink
3. Filter on source contradicts F
   $\Rightarrow$ replace M by empty relation and stop
4. Let F' be equal to F updated with information from other sources
5. Filter on source contradicts F'
   $\Rightarrow$ replace M by empty relation and stop
6. Filter on source implied by F
   $\Rightarrow$ stop (no parametrization required)
7. Filter on source implied by F'
   $\Rightarrow$ parametrize D and stop
8. Parametrize M

Additional Constraints on Parameters

- Some source iterations are definitely executed
  $\Rightarrow \lambda$ no later than definitely executed iterations

Additional Constraints on Parameters

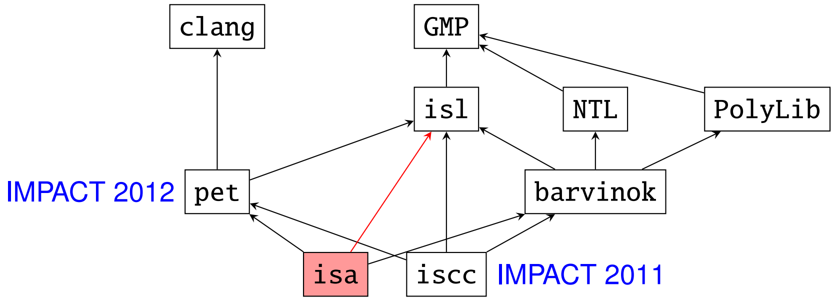
- Some source iterations are definitely executed
  $\Rightarrow \lambda$ no later than definitely executed iterations
- Eliminate (some) conflicts with other parameters

```
   state = 0;
   while (1) {
        sample = radioFrontend();
        if (t(state)) {
D:           state = detect(sample);
        } else {
C:           decode(sample, &state, &value0);
             value1 = processSample0(value0);
             processSample1(value1);
        }
   }
```

$\Rightarrow \lambda_0^{C}(i)$ and $\lambda_0^{D}(i)$ cannot both be smaller than $i - 1$

Outline

Interaction with Libraries



`isl`: manipulates parametric affine sets and relations

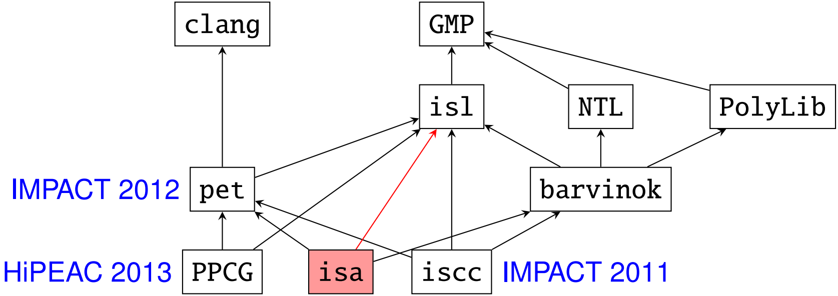
`barvinok`: counts elements in parametric affine sets and relations

`pet`: extracts polyhedral model from clang AST

`isa`: prototype tool set including

- derivation of process networks (with On Demand Parametric ADA)
- equivalence checker

Interaction with Libraries



`isl`: manipulates parametric affine sets and relations

`barvinok`: counts elements in parametric affine sets and relations

`pet`: extracts polyhedral model from clang AST

`isa`: prototype tool set including

- derivation of process networks (with On Demand Parametric ADA)
- equivalence checker

`PPCG`: Polyhedral Parallel Code Generator

Related Work

- Fuzzy Array Dataflow Analysis
  ⇒ only known publicly available implementation: `fadatool`
- Pugh et al. (1994) and Maslov (1995) produce approximate results
- Collard et al. (1999)
  - handle unstructured programs
  - only collect constraints
  - assume `Omega` can solve the constraints, but it cannot

Outline

Experimental Results

input	da			fadatool			fadatool -s		
	time	p	d	time	p	l	time	p	l
Example from paper	0.01s	0	5	0.01s	6	6	0.01s	6	6
Example from slides	0.01s	4	9	0.01s	6	16	incorrect		
fuzzy4	0.06s	3	9	0.02s	4	9	0.01s	0	9
for1	0.02s	2	3	0.01s	4	46	0.02s	2	3
for2	0.03s	2	3	0.09s	12	5k	0.04s	4	3
for3	0.04s	2	3	42s	24	1M	0.08s	6	3
for4	0.06s	2	3				0.16s	8	3
for5	0.08s	2	3				0.25s	10	3
for6	0.14s	2	3				0.42s	12	3
cascade_if1	0.02s	2	3	0.01s	2	4	0.01s	2	4
cascade_if2	0.02s	2	10	0.02s	4	52	0.02s	2	8
cascade_if3	0.03s	2	22	0.03	6	723	0.36s	3	16
cascade_if4	0.02s	2	10	0.17s	8	9k	1m	4	28
while1	0.01s	0	4	0.00s	1	4	0.01s	0	4
while2	0.03s	3	4	0.01s	5	6	incorrect		
if_var	0.03s	4	3	0.01s	2	8	0.01s	2	4
if_while	0.04s	2	14	0.01s	5	58	0.02s	4	58
if2	0.02s	2	2	0.46s	12	29k	0.04s	4	2

Experimental Results

input	da			fadatool			fadatool -s		
	time	p	d	time	p	l	time	p	l
Example from paper	0.01s	0	5	0.01s	6	6	0.01s	6	6
Example from slides	0.01s	4	9	0.01s	6	16	incorrect		
fuzzy4	0.06s	3	9	0.02s	4	9	0.01s	0	9
for1	0.02s	2	3	0.01s	4	46	0.02s	2	3
for2	0.03s	2	3	0.09s	12	5k	0.04s	4	3
for3	0.04s	2	3	42s	24	1M	0.08s	6	3
for4	0.06s	2	3				0.16s	8	3
for5	0.08s	2	3				0.25s	10	3
for6	0.14s	2	3				0.42s	12	3
cascade_if1	0.02s	2	3	0.01s	2	4	0.01s	2	4
cascade_if2	0.02s	2	10	0.02s	4	52	0.02s	2	8
cascade_if3	0.03s	2	22	0.03	6	723	0.36s	3	16
cascade_if4	0.02s	2	10	0.17s	8	9k	1m	4	28
while1	0.01s	0	4	0.00s	1	4	0.01s	0	4
while2	0.03s	3	4	0.01s	5	6	incorrect		
if_var	0.03s	4	3	0.01s	2	8	0.01s	2	4
if_while	0.04s	2	14	0.01s	5	58	0.02s	4	58
if2	0.02s	2	2	0.46s	12	29k	0.04s	4	2

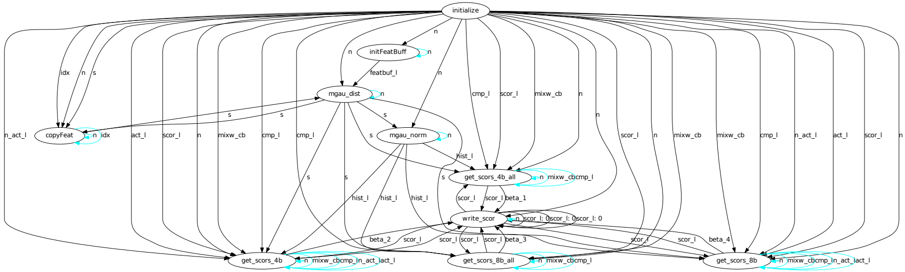
Experimental Results

input	da			fadatool			fadatool -s		
	time	p	d	time	p	l	time	p	l
Example from paper	0.01s	0	5	0.01s	6	6	0.01s	6	6
Example from slides	0.01s	4	9	0.01s	6	16	incorrect		
fuzzy4	0.06s	3	9	0.02s	4	9	0.01s	0	9
for1	0.02s	2	3	0.01s	4	46	0.02s	2	3
for2	0.03s	2	3	0.09s	12	5k	0.04s	4	3
for3	0.04s	2	3	42s	24	1M	0.08s	6	3
for4	0.06s	2	3				0.16s	8	3
for5	0.08s	2	3				0.25s	10	3
for6	0.14s	2	3				0.42s	12	3
cascade_if1	0.02s	2	3	0.01s	2	4	0.01s	2	4
cascade_if2	0.02s	2	10	0.02s	4	52	0.02s	2	8
cascade_if3	0.03s	2	22	0.03	6	723	0.36s	3	16
cascade_if4	0.02s	2	10	0.17s	8	9k	1m	4	28
while1	0.01s	0	4	0.00s	1	4	0.01s	0	4
while2	0.03s	3	4	0.01s	5	6	incorrect		
if_var	0.03s	4	3	0.01s	2	8	0.01s	2	4
if_while	0.04s	2	14	0.01s	5	58	0.02s	4	58
if2	0.02s	2	2	0.46s	12	29k	0.04s	4	2

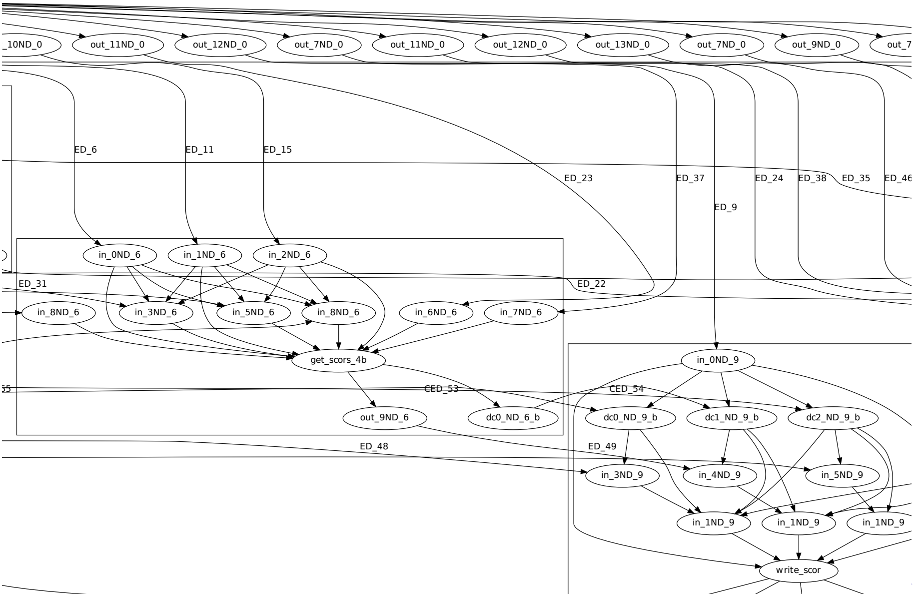
Larger Example — Input

```
for (j = 1; j <= frame; j++) {
  initialize(frame, n_act, &scor, &act, &ps, cmp,
             &s, &n, &idx, &mixw_cb, &cmp_l, &n_act_l, &act_l, &scor_l)
  for (i = 0; i < n; ++i) {
    initFeatBuff(i, &feat_buff, &featbuf_l);
    copyFeat(&s, frame, i, idx, &s);
    mgau_dist(&s, frame, i, &featbuf_l, &s);
    hist_l = mgau_norm(&s, frame, i);
    if (mixw_cb >= 1) {
      if (cmp_l >= 1)
        get_scors_4b_all(&s, i, hist_l, &scor_l, &scor_l);
      else
        get_scors_4b(&s, i, hist_l, n_act_l, &act_l, &scor_l, &scor_l);
    } else {
      if (cmp_l >= 1)
        get_scors_8b_all(&s, i, hist_l, &scor_l, &scor_l);
      else
        get_scors_8b(&s, i, hist_l, n_act_l, &act_l, &scor_l, &scor_l);
    }
    write_scor(&scor_l, &scor_l);
  }
}
```

Larger Example — Dataflow Graph



Larger Example — (Partial) Process Network



Outline

Conclusion

Conclusions

- Dynamic behavior represented using “filters”
- Exact, possibly parametric, dataflow analysis
- Prototype implementation in `isa`
- Similar to FADA, but
  - Parameters have a different meaning
  - Effect analyzed before parameters are introduced
  - All computations are performed directly on affine sets and maps

Future work

- Tighter integration into `pet`