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

1 Motivation
   - General Motivation
   - Our Motivation

2 Array Dataflow Analysis
   - Standard
   - Fuzzy
   - On Demand Parametric

3 Dynamic Conditions

4 Parametrization
   - Overview
   - Representation
   - Introduction
   - Additional Constraints

5 Related Work

6 Experimental Results

7 Conclusion
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
  - statement → process
  - flow dependence → communication channel

  ⇒ 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

  \[
  \text{statement} \quad \rightarrow \quad \text{process} \\
  \text{flow dependence} \quad \rightarrow \quad \text{communication channel}
  \]

  \[\Rightarrow \text{requires dataflow analysis}\]

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

Example:

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

```c
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

```c
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

```c
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

1 Motivation
   - General Motivation
   - Our Motivation

2 Array Dataflow Analysis
   - Standard
   - Fuzzy
   - On Demand Parametric

3 Dynamic Conditions

4 Parametrization
   - Overview
   - Representation
   - Introduction
   - Additional Constraints

5 Related Work

6 Experimental Results

7 Conclusion
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

```c
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

*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

```c
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:

\[ A1 := \{ N \} \rightarrow \{ F[i,j] \rightarrow a[i+j] : 0 \leq i < N \text{ and } 0 \leq j < N - i \} \]

\[ A2 := \{ N \} \rightarrow \{ W[i] \rightarrow a[i] : 0 \leq i < N \} \]

Map to all writes:

\[ R := A2 \cdot (A1^{-1}) \]

\[ \{ N \} \rightarrow \{ W[i] \rightarrow F[i',i-i'] : 0 \leq i,i' < N \text{ and } i' \leq i \} \]

Last write:

\[ \text{lexmax } R; \]

\[ \{ N \} \rightarrow \{ W[i] \rightarrow F[i,0] : 0 \leq i < N \} \]

In general: impose lexicographical order on shared iterators
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

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

```c
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

\[
\begin{array}{l}
\text{for } (i = 0; i < N; ++i) \\
\quad \text{for } (j = 0; j < N - i; ++j) \\
F: \quad a[i+j] = f(a[i+j]); \\
\text{for } (i = 0; i < N; ++i) \\
W: \quad \text{Write}(a[i]);
\end{array}
\]

Access relations:

\[
\begin{align*}
A1:=[N] & \rightarrow \{ F[i,j] \rightarrow a[i+j] : 0 \leq i < N \text{ and } 0 \leq j < N - i \} \\
A2:=[N] & \rightarrow \{ W[i] \rightarrow a[i] : 0 \leq i < N \}
\end{align*}
\]

Map to all writes: \( R := A2 \cdot (A1^{-1}) \);

\[
\begin{align*}
[N] & \rightarrow \{ W[i] \rightarrow F[i',i-i'] : 0 \leq i,i' < N \text{ and } i' \leq i \}
\end{align*}
\]
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

\[
\text{for } (i = 0; i < N; ++i) \\
\text{for } (j = 0; j < N - i; ++j) \\
F: \quad a[i+j] = f(a[i+j]); \\
\text{for } (i = 0; i < N; ++i) \\
W: \quad \text{Write}(a[i]);
\]

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

Map to all writes: \( R := A2 \cdot (A1^{-1}) \);
\[
[N] \to \{ W[i] -> F[i',i-i'] : 0 <= i,i' < N \text{ and } i'<= i \}
\]

Last write: \( \text{lexmax } R; \# [N] -> \{ W[i] -> F[i,0] : 0 <= i < N \} \)
### 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**

```plaintext
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**: 
  
  $$A1:=[N] \rightarrow \{ F[i,j] -> a[i+j] : 0 \leq i < N \text{ and } 0 \leq j < N - i \}$$

- **A2**: 
  
  $$A2:=[N] \rightarrow \{ W[i] -> a[i] : 0 \leq i < N \}$$

**Map to all writes:** 

$$R := A2 \cdot (A1^{-1})$$

- **Last write:** 
  
  $$\text{lexmax } R; \#\: [N] \rightarrow \{ W[i] -> F[i,0] : 0 \leq i < N \}$$

**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: 
  \[ D_{C,P}^{\text{mem}} = (A_P^{-1} \circ A_C) \cap B_C^P \]
  ("memory based dependences"; \( B_C^P \): \( P \) executed before \( C \))
- Source may already be known for some sink iterations
  \( \Rightarrow \) compute partial lexicographical maximum
  \[ (U', D) = \text{lexmax}_U M \]

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

Note: here, dependence relations map sink iterations to source iterations
Fuzzy Array Dataflow Analysis

- Introduces parameters for each $\text{lexmax}$ involving dynamic behavior
- Parameters represent dynamic solution of $\text{lexmax}$ operation
- Derives properties on parameters after dataflow analysis (using resolution)
Fuzzy Array Dataflow Analysis

- Introduces parameters for each \( \text{lexmax} \) involving dynamic behavior
- Parameters represent dynamic solution of \( \text{lexmax} \) 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 $\text{lex}_{\text{max}}$ involving dynamic behavior
- Parameters represent dynamic solution of $\text{lex}_{\text{max}}$ 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
⇒ 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

1 Motivation
   - General Motivation
   - Our Motivation

2 Array Dataflow Analysis
   - Standard
   - Fuzzy
   - On Demand Parametric

3 Dynamic Conditions

4 Parametrization
   - Overview
   - Representation
   - Introduction
   - Additional Constraints

5 Related Work

6 Experimental Results

7 Conclusion
Representing Generic Dynamic Conditions

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

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

Dynamic condition \( t(state) \) represented by \textit{filter}

- Filter access relation(s):
  - access to (virtual) array representing condition
    \[
    \{ D(i) \rightarrow (S_0(i) \rightarrow t_0(i)) \}
    \]

- Filter value relation:
  - values of filter array elements for which statement is executed
    \[
    \{ D(i) \rightarrow (1) \mid i \geq 0 \}
    \]
Representing Generic Dynamic Conditions

```plaintext
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
    \[
    \{ D(i) \rightarrow (S_0(i) \rightarrow t_0(i)) \}
    \]

- **Filter value relation:**
  - values of filter array elements for which statement is executed
    \[
    \{ D(i) \rightarrow (1) \mid i \geq 0 \}
    \]
Representing Generic Dynamic Conditions

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

Dynamic condition $t(state)$ represented by filter

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

  \[
  \{ D(i) \rightarrow (S_0(i) \rightarrow [t_0(i)]) \} 
  \]

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

  \[
  \{ D(i) \rightarrow (1) \mid i \geq 0 \} 
  \]
Representing Generic Dynamic Conditions

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

Dynamic condition \( t(\text{state}) \) represented by filter

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

\[
\{ D(i) \rightarrow (S_0(i) \rightarrow t_0(i)) \}
\]

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

\[
\{ D(i) \rightarrow (1) \mid i \geq 0 \}
\]
Representing Generic Dynamic Conditions

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

Dynamic condition \( t(state) \) represented by \textit{filter}

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

  \[
  \{ D(i) \rightarrow (S_0(i) \rightarrow t_0(i)) \}
  \]

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

  \[
  \{ D(i) \rightarrow (1) \mid i \geq 0 \}
  \]
Representing Locally Static Affine Conditions

N1: \( n = f(); \)

\[
\begin{align*}
\text{for (int } k = 0; k < 100; ++k) \{ \\
\text{M: } m = g(); \\
\text{for (int } i = 0; i < m; ++i) \\
\text{for (int } j = 0; j < n; ++j) \\
\text{A: } a[j][i] = g(); \\
\text{N2: } n = f();
\}
\end{align*}
\]

Values of \( m \) and \( n \) not changed inside \( i \) and \( j \) loops
\Rightarrow \text{locally static affine loop conditions}
Representing Locally Static Affine Conditions

N1: \( n = f(); \)

\[
\text{for (int } k = 0; k < 100; ++k) \{
\]

M: \( m = g(); \)

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

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:
  \[
  \{ A(k, i, j) \rightarrow (M(k) \rightarrow m()) \} \\
  \{ A(0, i, j) \rightarrow (N1()) \rightarrow n()) \} \cup \{ A(k, i, j) \rightarrow (N2(k - 1) \rightarrow n()) | k \geq 1 \} \\
  \]

- Filter value relation:
  \[
  \{ A(k, i, j) \rightarrow (m, n) | 0 \leq k \leq 99 \land 0 \leq i < m \land 0 \leq j < n \} \\
  \]

Note: filter access relations exploit (static) dataflow analysis on \( m \) and \( n \).
Outline

1. Motivation
   - General Motivation
   - Our Motivation

2. Array Dataflow Analysis
   - Standard
   - Fuzzy
   - On Demand Parametric

3. Dynamic Conditions

4. Parametrization
   - Overview
   - Representation
   - Introduction
   - Additional Constraints

5. Related Work

6. Experimental Results

7. Conclusion
Overview

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

state = 0;

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

Memory based dependences:

\[ D \text{ mem } C, P = \{ S_0(i) \rightarrow D(i') | 0 \leq i' < i \} \]

At \( \ell = 1 \):
\[ M = D \text{ mem } C, P \cap \{ S_0(i) \rightarrow D(i) \} = \emptyset \]

At \( \ell = 0 \):
\[ M = \{ S_0(i) \rightarrow D(i') | 0 \leq i' < i \} \]

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

\( \Rightarrow \) parametrization required
Parametrization

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

Memory based dependences:

- **D** \( \text{mem} \) \( C \)
- \( P = \{ \text{S}0(i) \rightarrow D(i') | 0 \leq i' < i \} \)

At \( \ell = 1 \):

- \( M = \text{D} \text{mem} \cap \{ \text{S}0(i) \rightarrow D(i) \} = \emptyset \)

At \( \ell = 0 \):

- \( M = \{ \text{S}0(i) \rightarrow D(i') | 0 \leq i' < i \} \)

Potential source \( D(i') \) may not have executed even if sink \( S0(i) \) is executed \( \Rightarrow \) parametrization required

- sink \( C \)
- potential source \( P \)
Parametrization

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

Memory based dependences: $D_{C,P}^{\text{mem}} = \{ S_0(i) \rightarrow D(i') | 0 \leq i' < i \}$

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

At $\ell = 0$: $M = \{ S_0(i) \rightarrow D(i') | 0 \leq i' < i \}$

Potential source $D(i')$ may not have executed even if sink $S_0(i)$ is executed
$\Rightarrow$ parametrization required
Parameter Representation

Original:

\[ M = \{ S_0(i) \rightarrow D(i') \mid 0 \leq i' < i \} \]

After parameter introduction:

\[ M' = \{ S_0(i) \rightarrow D(\lambda^P_C(i)) \mid 0 \leq \lambda^P_C(i) < i \land \beta^P_C(i) = 1 \} \]

\[ \Rightarrow \text{lexmax } M' = M' \]
Parameter Representation

Original:

\[ M = \{ S_0(i) \rightarrow D(i') \mid 0 \leq i' < i \} \]

After parameter introduction:

\[ M' = \{ S_0(i) \rightarrow D(\lambda_C^P(i)) \mid 0 \leq \lambda_C^P(i) < i \land \beta_C^P(i) = 1 \} \]

\[ \Rightarrow \text{lexmax } M' = M' \]

Meaning of the parameters:

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

Note: FADA introduces separate set of parameters for each lexmax

Note: \( \lambda_C^P(k) \) and \( \beta_C^P(k) \) depend on \( k \), but dependence can be kept implicit

\[ \Rightarrow \lambda_C^P \text{ and } \beta_C^P \]
Introducing as few Parameters as possible

In principle, the number of elements in $\lambda$ 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 $\lambda$ 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

$$\beta_k = 1 \rightarrow \sigma \geq \ell$$

$$\beta_k = 0 \rightarrow \sigma < \ell$$

When moving to $\lambda_{\ell-1}$, introduce additional parameter $\lambda_{\ell-1}$ (if needed).

Make implicit equality explicit at the end of the dataflow analysis.

$$\sigma \geq \ell \leq \beta_k$$

$$\sigma < \ell \leq \beta_k$$

($\ell \leq D_k$: smallest $\ell$ for which parametrization was applied)

$\lambda_k$ and $\beta_k$ now refer to last execution of $D_k$ ($D_k$: result of projecting out parameters from final dataflow relation).
Introducing as few Parameters as possible

In principle, the number of elements in $\lambda$ 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

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

$$M = \{ B() \rightarrow A(i,j) \mid 0 \leq i, j < 100 \}$$
Introducing as few Parameters as possible

In principle, the number of elements in $\lambda$ 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

```c
for (i = 0; i < 100; ++i)
    if (t())
        for (j = 0; j < 100; ++j)
            a = t();
```

A: $b = a$;

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

$$M' = \{ B() \rightarrow A(\lambda_0,j) \mid 0 \leq \lambda_0, j < 100 \& \beta = 1 \}$$
Introducing as few Parameters as possible

In principle, the number of elements in $\lambda$ 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

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

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

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

$$\text{lexmax} \ M' = \{ B() \to A(\lambda_0, 99) \mid 0 \leq \lambda_0 < 100 \land \beta = 1 \}$$
Introducing as few Parameters as possible

In principle, the number of elements in $\lambda$ 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

Dimensions that can only attain a single value

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

\[
D_{H,A}^{\text{mem}} = \{ H(k, i, j) \rightarrow A(k', i, j) \mid k' \leq k \}
\]

\[
\lambda_1(k, i, j) = i \\
\lambda_2(k, i, j) = j
\]

\[
\Rightarrow \text{no need to introduce } \lambda_1 \text{ and } \lambda_2
\]
Introducing as few Parameters as possible

In principle, the number of elements in $\lambda$ 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 $\lambda$ 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 $\ell$
Introducing as few Parameters as possible

Dimensions before $\ell$

```c
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 $\ell = 1$:

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

$\Rightarrow$ no need to introduce $\lambda_0$ (yet) at $\ell = 1$
Introducing as few Parameters as possible

Dimensions before $\ell$

```c
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 $\ell = 1$:

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

$\Rightarrow$ no need to introduce $\lambda_0$ (yet) at $\ell = 1$

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

$\Rightarrow$ no need to consider $\ell = 0$ and $\lambda_0$ not needed at all
Introducing as few Parameters as possible

In principle, the number of elements in $\lambda$ 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 $\ell$
Introducing as few Parameters as possible

In principle, the number of elements in $\lambda$ 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 $\ell$

$\Rightarrow$ replace $\beta$ by $\sigma$: the number of implicitly equal shared iterators

\[
\begin{align*}
\beta &= 1 & \rightarrow & & \sigma \geq \ell \\
\beta &= 0 & \rightarrow & & \sigma < \ell
\end{align*}
\]
Introducing as few Parameters as possible

In principle, the number of elements in $\lambda$ is equal to the number of iterators $\lambda$. 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 $\ell$

\[ \Rightarrow \text{replace } \beta \text{ by } \sigma: \text{the number of implicitly equal shared iterators} \]

\[ \begin{align*}
\beta = 1 & \rightarrow \sigma \geq \ell \\
\beta = 0 & \rightarrow \sigma < \ell
\end{align*} \]

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

\[ \begin{align*}
\sigma \geq \ell_{\leq} & \rightarrow \beta = 1 \\
\sigma < \ell_{\leq} & \rightarrow \beta = 0
\end{align*} \]

($\ell_{\leq}$: smallest $\ell$ for which parametrization was applied)

$\lambda(k)$ and $\beta(k)$ now refer to last execution of $\overline{D}(k)$

($\overline{D}$: 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

\[
(U', D) = \text{\text{lexmax}} \frac{M}{U}
\]
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)

Computing

$$(U', D) = \text{lexmax}_U M$$
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
   \[ \Rightarrow \text{stop (no parametrization required)} \]

2. Let \( F \) be the filter on the sink
3. Filter on source contradicts \( F \)
   \[ \Rightarrow \text{replace } M \text{ by empty relation and stop} \]

Computing

\[ (U', D) = \text{lexmax}_U M \]
Filter on source contradicts $F$

\[ \ell = 1 \]

- Potential source filter access relation
  \[ \{ H(i) \rightarrow (N(i) \rightarrow n) \} \]

- Potential source filter value relation
  \[ \{ H(i) \rightarrow (n) \mid i \geq 0 \land n < 100 \} \]

- Sink filter access relation
  \[ \{ T(i) \rightarrow (N(i) \rightarrow n) \} \]

- Sink filter value relation
  \[ \{ T(i) \rightarrow (n) \mid i \geq 0 \land n > 200 \} \]
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 filter access relation
  \[ \{ H(i) \rightarrow (N(i) \rightarrow n) \} \]
- Potential source filter value relation
  \[ \{ H(i) \rightarrow (n) \mid i \geq 0 \land n < 100 \} \]
- Sink filter access relation
  \[ \{ T(i) \rightarrow (N(i) \rightarrow n) \} \]
- Sink filter value relation
  \[ \{ T(i) \rightarrow (n) \mid i \geq 0 \land n > 200 \} \]
Filter on source contradicts $F$

\[
\ell = 1
\]

same filter element

\[
\begin{aligned}
\textbf{while } & (1) \{ \\
\text{N: } & n = f(); \\
a & = g(); \\
\text{if } & (n < 100) \\
\text{H: } & a = h(); \\
\text{if } & (n > 200) \\
\text{T: } & t(a); \\
\}
\end{aligned}
\]

potential source

\[
\begin{aligned}
\text{Potential source filter access relation} & \quad \{ H(i) \rightarrow (N(i) \rightarrow n) \} \\
\text{Potential source filter value relation} & \quad \{ H(i) \rightarrow (n) \mid i \geq 0 \land n < 100 \}
\end{aligned}
\]

sink

\[
\begin{aligned}
\text{Sink filter access relation} & \quad \{ T(i) \rightarrow (N(i) \rightarrow n) \} \\
\text{Sink filter value relation} & \quad \{ T(i) \rightarrow (n) \mid i \geq 0 \land n > 200 \}
\end{aligned}
\]

contradiction
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
   \[ \Rightarrow \text{stop (no parametrization required)} \]

2. Let \( F \) be the filter on the sink

3. Filter on source contradicts \( F \)
   \[ \Rightarrow \text{replace } M \text{ by empty relation and stop} \]

Computing

\[
(U', D) = \text{lexmax}_U M
\]
When to Introduce Parameters

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

Computing

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

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 $F'$

\begin{align*}
N: \quad & n = f(); \\
    & \text{if} \ (n < 100) \\
H: \quad & a = h(); \\
    & \text{if} \ (n < 200) \\
H2: \quad & a = h2(); \\
T: \quad & t(a); \\
\text{sink}
\end{align*}

potential source

\[
\text{lexmax } M = \\{ T() \to H() \}
\]

\[
U = \{ T() \mid \sigma^H2 < 0 \}
\]
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**

\[ \text{lexmax } M \]

\[ U = \{ T() \rightarrow H() \} \]

\[ U = \{ T() \mid \sigma^{H2} < 0 \} \]

**sink**

**H2 not executed**
Filter on source contradicts $F'$

N: \hspace{1em} n = f();
    \hspace{1em} \textbf{if} (n < 100)
H: \hspace{1em} a = h();
    \hspace{1em} \textbf{if} (n < 200)
H2: \hspace{1em} a = h2();
T: \hspace{1em} t(a);
}

Updated sink filter access relation

\[ \{ T(i) \rightarrow (N(i) \rightarrow n) \} \]

Updated sink filter value relation

\[ \{ T(i) \rightarrow (n) \mid i \geq 0 \land n \geq 200 \} \]
When to Introduce Parameters

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

Computing

$(U', D) = \text{lexmax } M_U$

1. No filter on source
   ⇒ stop (no parametrization required)

2. Let $F$ be the filter on the sink
3. Filter on source contradicts $F$
   ⇒ 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 $F'$
   ⇒ 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 \( M \)

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)

Computing
\[
(U', D) = \operatorname{lexmax}_U M
\]
Filter on source implied by $F$

\[ \ell = 1 \]

- Potential source filter access relation
  \[ \left\{ H(i) \rightarrow (N(i) \rightarrow n) \right\} \]

- Potential source filter value relation
  \[ \left\{ H(i) \rightarrow (n) \mid i \geq 0 \land n < 200 \right\} \]

- Sink filter access relation
  \[ \left\{ T(i) \rightarrow (N(i) \rightarrow n) \right\} \]

- Sink filter value relation
  \[ \left\{ T(i) \rightarrow (n) \mid i \geq 0 \land n < 100 \right\} \]
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

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)

Computing

$$(U', D) = \text{lexmax}_U M$$
When to Introduce Parameters

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

Computing

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

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 \( F' \)

\[
N: \quad n = f();
\]
\[
\text{if } (n < 200)
\]
\[
H: \quad a = h();
\]
\[
\text{if } (n > 100)
\]
\[
H2: \quad a = h2();
\]
\[
T: \quad t(a);
\]
\[
\}
\]

- Updated sink filter access relation

\[
\left\{ T(i) \rightarrow (N(i) \rightarrow n) \right\}
\]

- Updated sink filter value relation

\[
\left\{ T(i) \rightarrow (n) \mid i \geq 0 \land n \leq 100 \right\}
\]
When to Introduce Parameters

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

Computing

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

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 $M$

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 $M$

Computing

$$(U', D) = \operatorname{lexmax}_U M$$
Additional Constraints on Parameters

- Some source iterations are definitely executed
  \[ \Rightarrow \lambda \text{ no later than definitely executed iterations} \]
Additional Constraints on Parameters

- Some source iterations are definitely executed
  \[ \Rightarrow \lambda \text{ no later than definitely executed iterations} \]
- Eliminate (some) conflicts with other parameters

\[
\text{state} = 0; \\
\textbf{while} (1) \{ \\
\quad \text{sample} = \text{radioFrontend}(); \\
\quad \textbf{if} (t(\text{state})) \{ \\
\quad \quad \text{D: } \text{state} = \text{detect}()(); \\
\quad \quad \} \textbf{else} \{ \\
\quad \quad \text{C: } \text{decode}(); \\
\quad \quad \quad \text{value1} = \text{processSample0}(); \\
\quad \quad \quad \text{processSample1}(); \\
\quad \quad \} \\
\}
\]

\[\Rightarrow \lambda^C_0(i) \text{ and } \lambda^D_0(i) \text{ cannot both be smaller than } i - 1\]
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
Interation 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
  $\Rightarrow$ 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 $\Omega$mega can solve the constraints, but it cannot
## Experimental Results

<table>
<thead>
<tr>
<th>input</th>
<th>da</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>time</td>
<td>p</td>
<td>d</td>
</tr>
<tr>
<td>Example from paper</td>
<td></td>
<td>0.01s</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>Example from slides</td>
<td></td>
<td>0.01s</td>
<td>4</td>
<td>9</td>
</tr>
<tr>
<td>fuzzy4</td>
<td></td>
<td>0.06s</td>
<td>3</td>
<td>9</td>
</tr>
<tr>
<td>for1</td>
<td></td>
<td>0.02s</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>for2</td>
<td></td>
<td>0.03s</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>for3</td>
<td></td>
<td>0.04s</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>for4</td>
<td></td>
<td>0.06s</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>for5</td>
<td></td>
<td>0.08s</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>for6</td>
<td></td>
<td>0.14s</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>cascade_if1</td>
<td></td>
<td>0.02s</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>cascade_if2</td>
<td></td>
<td>0.02s</td>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>cascade_if3</td>
<td></td>
<td>0.03s</td>
<td>2</td>
<td>22</td>
</tr>
<tr>
<td>cascade_if4</td>
<td></td>
<td>0.02s</td>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>while1</td>
<td></td>
<td>0.01s</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>while2</td>
<td></td>
<td>0.03s</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>if_var</td>
<td></td>
<td>0.03s</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>if_while</td>
<td></td>
<td>0.04s</td>
<td>2</td>
<td>14</td>
</tr>
<tr>
<td>if2</td>
<td></td>
<td>0.02s</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

|            | fadatool |          |          |          |
|            |          | time     | p        | l        |
| Example from paper        | 0.01s    | 6        | 6        |
| Example from slides       | 0.01s    | 6        | 16       |
| fuzzy4                    | 0.02s    | 4        | 9        |
| for1                      | 0.01s    | 4        | 46       |
| for2                      | 0.09s    | 12       | 5k       |
| for3                      | 42s      | 24       | 1M       |
| for4                      | 0.06s    | 2        | 3        |
| for5                      | 0.04s    | 4        | 3        |
| for6                      | 0.16s    | 8        | 3        |
| cascade_if1               | 0.01s    | 2        | 4        |
| cascade_if2               | 0.02s    | 4        | 52       |
| cascade_if3               | 0.03     | 6        | 723      |
| cascade_if4               | 0.17s    | 8        | 9k       |
| while1                    | 0.00s    | 1        | 4        |
| while2                    | 0.01s    | 5        | 6        |
| if_var                    | 0.01s    | 2        | 8        |
| if_while                  | 0.01s    | 5        | 58       |
| if2                       | 0.46s    | 12       | 29k      |

|            | fadatool -s |          |          |          |
|            |             | time     | p        | l        |
| Example from paper        | 0.01s    | 6        | 6        |
| Example from slides       | incorrect |
| fuzzy4                    | 0.01s    | 0        | 9        |
| for1                      | 0.02s    | 2        | 3        |
| for2                      | 0.04s    | 4        | 3        |
| for3                      | 0.08s    | 6        | 3        |
| for4                      | 0.16s    | 8        | 3        |
| for5                      | 0.25s    | 10       | 3        |
| for6                      | 0.42s    | 12       | 3        |
| cascade_if1               | 0.01s    | 2        | 4        |
| cascade_if2               | 0.02s    | 2        | 8        |
| cascade_if3               | 0.36s    | 3        | 16       |
| cascade_if4               | 1m       | 4        | 28       |
| while1                    | 0.01s    | 0        | 4        |
| while2                    | incorrect |
| if_var                    | 0.01s    | 2        | 4        |
| if_while                  | 0.02s    | 4        | 58       |
| if2                       | 0.04s    | 4        | 2        |
### Experimental Results

<table>
<thead>
<tr>
<th>input</th>
<th>da</th>
<th>fadatool</th>
<th>fadatool -s</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>time</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>p</td>
<td>d</td>
</tr>
<tr>
<td>Example from paper</td>
<td>0.01s</td>
<td>0</td>
<td>5</td>
</tr>
<tr>
<td>Example from slides</td>
<td>0.01s</td>
<td>4</td>
<td>9</td>
</tr>
<tr>
<td>fuzzy4</td>
<td>0.06s</td>
<td>3</td>
<td>9</td>
</tr>
<tr>
<td>for1</td>
<td>0.02s</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>for2</td>
<td>0.03s</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>for3</td>
<td>0.04s</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>for4</td>
<td>0.06s</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>for5</td>
<td>0.08s</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>for6</td>
<td>0.14s</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>cascade_if1</td>
<td>0.02s</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>cascade_if2</td>
<td>0.02s</td>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>cascade_if3</td>
<td>0.03s</td>
<td>2</td>
<td>22</td>
</tr>
<tr>
<td>cascade_if4</td>
<td>0.02s</td>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>while1</td>
<td>0.01s</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>while2</td>
<td>0.03s</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>if_var</td>
<td>0.03s</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>if_while</td>
<td>0.04s</td>
<td>2</td>
<td>14</td>
</tr>
<tr>
<td>if2</td>
<td>0.02s</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>input</td>
<td>da</td>
<td>fadatool</td>
<td>fadatool -s</td>
</tr>
<tr>
<td>-------------------------------</td>
<td>----</td>
<td>----------</td>
<td>-------------</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>time  p  d</td>
<td>time  p  l</td>
</tr>
<tr>
<td>Example from paper</td>
<td></td>
<td>0.01s  0  5</td>
<td>0.01s  6  6</td>
</tr>
<tr>
<td>Example from slides</td>
<td></td>
<td>0.01s  4  9</td>
<td>0.01s  6  16</td>
</tr>
<tr>
<td>fuzzy4</td>
<td></td>
<td>0.06s  3  9</td>
<td>0.02s  4  9</td>
</tr>
<tr>
<td>for1</td>
<td></td>
<td>0.02s  2  3</td>
<td>0.01s  4  46</td>
</tr>
<tr>
<td>for2</td>
<td></td>
<td>0.03s  2  3</td>
<td>0.09s  12 5k</td>
</tr>
<tr>
<td>for3</td>
<td></td>
<td>0.04s  2  3</td>
<td>42s  24 1M</td>
</tr>
<tr>
<td>for4</td>
<td></td>
<td>0.06s  2  3</td>
<td></td>
</tr>
<tr>
<td>for5</td>
<td></td>
<td>0.08s  2  3</td>
<td></td>
</tr>
<tr>
<td>for6</td>
<td></td>
<td>0.14s  2  3</td>
<td></td>
</tr>
<tr>
<td>cascade_if1</td>
<td></td>
<td>0.02s  2  3</td>
<td>0.01s  2  4</td>
</tr>
<tr>
<td>cascade_if2</td>
<td></td>
<td>0.02s  2  10</td>
<td>0.02s  4  52</td>
</tr>
<tr>
<td>cascade_if3</td>
<td></td>
<td>0.03s  2  22</td>
<td>0.03  6 723</td>
</tr>
<tr>
<td>cascade_if4</td>
<td></td>
<td>0.02s  2  10</td>
<td>0.17s  8 9k</td>
</tr>
<tr>
<td>while1</td>
<td></td>
<td>0.01s  0  4</td>
<td>0.00s  1  4</td>
</tr>
<tr>
<td>while2</td>
<td></td>
<td>0.03s  3  4</td>
<td>0.01s  5  6</td>
</tr>
<tr>
<td>if_var</td>
<td></td>
<td>0.03s  4  3</td>
<td>0.01s  2  8</td>
</tr>
<tr>
<td>if_while</td>
<td></td>
<td>0.04s  2  14</td>
<td>0.01s  5  58</td>
</tr>
<tr>
<td>if2</td>
<td></td>
<td>0.02s  2  2</td>
<td>0.46s  12 29k</td>
</tr>
</tbody>
</table>
Larger Example — Input

```c
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

[Diagram of a process network with various nodes and edges labeled with operations such as ED, get_scors_4b, and write_scor.]
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