Static Analysis of OpenStream Programs

Using polyhedral techniques to analyze interesting language subsets

Alain Darte
With Albert Cohen and Paul Feautrier

CNRS, Compsys
Laboratoire de l'Informatique du Parallélisme
École normale supérieure de Lyon

IMPACT'16
6th Int. Workshop on Polyhedral Compilation Techniques
Tuesday, January 19, 2016. Prague, Czech Republic.

Partly supported by the ManycoreLabs project PIA-6394
led by the manycore company Kalray.

Parallel languages, runtime execution, and static analysis

Solution(s) for high-level parallel programming?

- Optimizations: static or dynamic?
- Specifications: language constructs or libraries?
- Expressiveness: deterministic (no data races) or deadlock-free?
- How to represent communications and memories? Concurrency?

Parallel languages, runtime execution, and static analysis

Solution(s) for high-level parallel programming?

- Optimizations: static or dynamic?
- Specifications: language constructs or libraries?
- Expressiveness: deterministic (no data races) or deadlock-free?
- How to represent communications and memories? Concurrency?

Endless list of approaches:

- "Lower"-level: MPI, CUDA, OpenCL, Lime, ...
- Runtime-based: Kaapi, StarPU (with task dep. as in OpenMP 4.0), TBB, ...
- (A)PGAS languages: Co-Array Fortran, UPC, Chapel, X10, ...
- "Dataflow" languages: KPN, SDF, CSDF, StreamIt, SigmaC, OpenStream, ...
- Many other types: OpenMP, StarSs, SAC, Concurrent Collections, Galois, ...

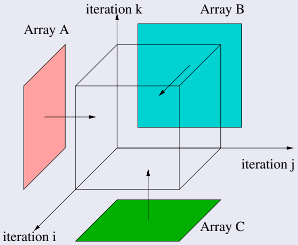
☛ Can static optimization help runtime optimizations?

Worst-case, liveness, deadlocks, races, buffer sizes, granularity, locality, ...

Multi-dimensional affine representation of loops and arrays

Matrix Multiply

```
int i,j,k;
for(i = 0; i < n; i++) {
   for(j = 0; j < n; j++) {
S:      C[i][j] = 0;
        for(k = 0; k < n; k++) {
T:          C[i][j] += A[i][k] * B[k][j];
        }
   }
}
```



Polyhedral Description — Omega/ISCC-like syntax

```
Domain := [n]->{S[i,j]: 0<=i,j<n; T[i,j,k]: 0<=i,j,k<n};

Read := [n]->{T[i,j,k]->A[i,k]; T[i,j,k]->B[k,j];
                T[i,j,k]->C[i,j]};

Write := [n]->{S[i,j]->C[i,j]; T[i,j,k]->C[i,j]};

Order := [n]->{S[i,j]->[i,j,0]; T[i,j,k]->[i,j,1,k]};
```

Triple interest of polyhedral model

Polyhedral "model", model of what?

- Specification model: affine loops, Alpha, CRP
- Provable techniques with some hypotheses: SCoP, approximations.
- Simplified form to prove hardnesss: NP-completeness, undecidability.

☛ Limits of automation often related to polyhedral model.

Triple interest of polyhedral model

Polyhedral "model", model of what?

- Specification model: affine loops, Alpha, CRP
- Provable techniques with some hypotheses: SCoP, approximations.
- Simplified form to prove hardnesss: NP-completeness, undecidability.

☛ Limits of automation often related to polyhedral model.

Principle: study a polyhedral subset of a specification/language.

- Uniform loops as simple cases to discuss NP-completeness.
- Polyhedral X10 (Yuki, Feautrier, Rajopadhye, Saraswat, PPoPP'13).
- Polyhedral OpenStream (Pop/Cohen CDDF + this paper).

☛ Part of an effort in extending (with new techniques) and expanding (with new applications) polyhedral compilation.

Analyzing X10 through a polyhedral fragment

X10 language developed at IBM, variant at Rice (V. Sarkar)

- PGAS (partitioned global address space) memory principle.
- Parallelism of threads: in particular keywords finish, async, clock.
- No deadlocks by construction but non-determinism is possible.

Polyhedral X10 Yuki, Feautrier, Rajopadhye, Saraswat (PPoPP 2013)

Can we analyze the code for data races?

```
finish {
  for(i in 0..n-1) {
    S1;
    async {
      S2;
    }
  }
}
```

```
clocked finish {
  for(i in 0..n-1) {
    S1; advance();
    clocked async {
      S2; advance();
    }
  }
}
```

Analyzing X10 through a polyhedral fragment

X10 language developed at IBM, variant at Rice (V. Sarkar)

- PGAS (partitioned global address space) memory principle.
- Parallelism of threads: in particular keywords finish, async, clock.
- No deadlocks by construction but non-determinism is possible.

Polyhedral X10 Yuki, Feautrier, Rajopadhye, Saraswat (PPoPP 2013)

Can we analyze the code for data races?

```
finish {
  for(i in 0..n-1) {
    S1;
    async {
      S2;
    }
  }
}
```

```
clocked finish {
  for(i in 0..n-1) {
    S1; advance();
    clocked async {
      S2; advance();
    }
  }
}
```

Yes. Similar to data-flow analysis, with partial order $\prec$ (incomplete lexicographic order).

Analyzing X10 through a polyhedral fragment

X10 language developed at IBM, variant at Rice (V. Sarkar)

- PGAS (partitioned global address space) memory principle.
- Parallelism of threads: in particular keywords finish, async, clock.
- No deadlocks by construction but non-determinism is possible.

Polyhedral X10 Yuki, Feautrier, Rajopadhye, Saraswat (PPoPP 2013)

Can we analyze the code for data races?

```
finish {
  for(i in 0..n-1) {
    S1;
    async {
      S2;
    }
  }
}
```

Yes. Similar to data-flow analysis, with partial order $\prec$ (incomplete lexicographic order).

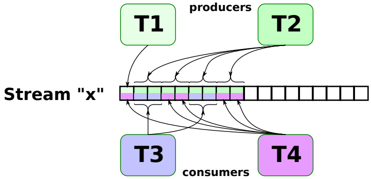
```
clocked finish {
  for(i in 0..n-1) {
    S1; advance();
    clocked async {
      S2; advance();
    }
  }
}
```

Undecidable. Partial order $\prec_c$ defined by $\vec{x} \prec_c \vec{y}$ iff $\vec{x} \prec \vec{y}$ or $\phi(\vec{x}) < \phi(\vec{y})$. $\phi(\vec{x}) = \#$ advances before (for $\prec$) $\vec{x}$.

Analyzing OpenStream through a polyhedral fragment

```
#pragma omp task output (x) // Task T1
x = ...;

for (i = 0; i < N; ++i) {
  int window_a[2], window_b[3];

  #pragma omp task output (x « window_a[2]) // Task T2
  window_a[0] = ...; window_a[1] = ...;

  if (i % 2) {
    #pragma omp task input (x » window_b[2]) // Task T3
    use (window_b[0], window_b[1]);
  }

  #pragma omp task input (x) // Task T4
  use (x);
}
```

(Pop, Cohen, 2011)

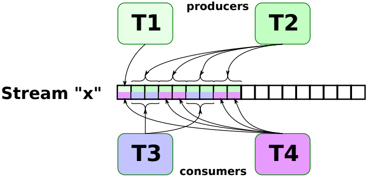


- Sequential control program for task creations ($\neq$ activations).
- Unlike KPN, streams with multiple inputs/outputs (but deterministic).

Analyzing OpenStream through a polyhedral fragment

```
#pragma omp task output (x) // Task T1
x = ...;

for (i = 0; i < N; ++i) {
  int window_a[2], window_b[3];

  #pragma omp task output (x « window_a[2]) // Task T2
  window_a[0] = ...; window_a[1] = ...;

  if (i % 2) {
    #pragma omp task input (x » window_b[2]) // Task T3
    use (window_b[0], window_b[1]);
  }

  #pragma omp task input (x) // Task T4
  use (x);
}
```

(Pop, Cohen, 2011)



- Sequential control program for task creations ($\neq$ activations).
- Unlike KPN, streams with multiple inputs/outputs (but deterministic).

- Reservation for reads/writes in streams with burst and horizon.
- Single assignment in streams (by construction) + dataflow semantics.
- The order of creations is the sequential order of the control program.

- Erbium runtime, optimizations of OpenStream explored by Pop, Miranda & Cohen. Motivates the analysis of a polyhedral fragment.

Some properties of polyhedral OpenStream

- Write/read access functions to streams are polynomials that can be expressed statically (loop counting: Ehrhart, Barvinok).

$$\text{Ex. for writes: } I_s(\vec{t}\,) = \sum_{\tau \in W_s} b_{\tau,s} \mathrm{Card}\{\vec{x} \in D_\tau \mid \vec{x} \prec_{\mathrm{lex}} \vec{t}\,\}$$

- Dependence analysis and scheduling are "feasible" with tools capable of handling polynomials. ☛ link with P. Feautrier's IMPACT'15 paper.

Some properties of polyhedral OpenStream

- Write/read access functions to streams are polynomials that can be expressed statically (loop counting: Ehrhart, Barvinok).

$$\text{Ex. for writes: } I_s(\vec{t}\,) = \sum_{\tau \in W_s} b_{\tau,s} \text{Card}\{\vec{x} \in D_\tau \mid \vec{x} \prec_{\text{lex}} \vec{t}\,\}$$

- Dependence analysis and scheduling are “feasible” with tools capable of handling polynomials. ☛ link with P. Feautrier's IMPACT'15 paper.
- Deadlocks do not depend on the execution order of tasks (as KPN).

Some properties of polyhedral OpenStream

- Write/read access functions to streams are polynomials that can be expressed statically (loop counting: Ehrhart, Barvinok).

$$\text{Ex. for writes: } I_s(\vec{t}\,) = \sum_{\tau \in W_s} b_{\tau,s} \text{Card}\{\vec{x} \in D_\tau \mid \vec{x} \prec_{\text{lex}} \vec{t}\,\}$$

- Dependence analysis and scheduling are "feasible" with tools capable of handling polynomials. ☛ link with P. Feautrier's IMPACT'15 paper.
- Deadlocks do not depend on the execution order of tasks (as KPN).
- If a schedule exists with bounded streams, such sizes can be enforced by blocking R/W, without creating deadlocks at runtime.
  - Buffer of size s: window of s live elements moving to increasing indices.

Some properties of polyhedral OpenStream

- Write/read access functions to streams are polynomials that can be expressed statically (loop counting: Ehrhart, Barvinok).

$$\text{Ex. for writes: } I_s(\vec{t}\,) = \sum_{\tau \in W_s} b_{\tau,s} \mathrm{Card}\{\vec{x} \in D_\tau \mid \vec{x} \prec_{\text{lex}} \vec{t}\,\}$$

- Dependence analysis and scheduling are "feasible" with tools capable of handling polynomials. ☛ link with P. Feautrier's IMPACT'15 paper.
- Deadlocks do not depend on the execution order of tasks (as KPN).
- If a schedule exists with bounded streams, such sizes can be enforced by blocking R/W, without creating deadlocks at runtime.
  - Buffer of size s: window of s live elements moving to increasing indices.
- Deadlock detection is undecidable (polynomials encoding as for X10).
  - With dependences only, where a read waits for its corresponding write.
  - Even if a read must wait for all writes with smaller indices ("Kahnian").
  - Even if writes must occur in increasing order of their indices ("causal").

First ingredient (Feautrier): build multivariate polynomials

$Q(x_1, \ldots, x_n)$: multivariate polynomial, nonnegative integer coefficients.

Write:

- $Q(x) = Q(x_1, x_r)$, x_1 first variable.
- $Q^1(x_1, x_r) = Q(x_1 + 1, x_r) - Q(x_1, x_r)$ (first difference)
  - ☛ smaller degree, still nonnegative integer coefficients.

First ingredient (Feautrier): build multivariate polynomials

$Q(x_1, \ldots, x_n)$: multivariate polynomial, nonnegative integer coefficients.

Write:

- $Q(x) = Q(x_1, x_r)$, x_1 first variable.
- $Q^1(x_1, x_r) = Q(x_1 + 1, x_r) - Q(x_1, x_r)$ (first difference)
  - ☛ smaller degree, still nonnegative integer coefficients.
  - ☛ Can compute $Q(x)$ with:

```
phi = Q(0,x_r);
for (i = 0; i < x; i++) {
  phi += Q1(i, x_r);
}
```

First ingredient (Feautrier): build multivariate polynomials

$Q(x_1, \ldots, x_n)$: multivariate polynomial, nonnegative integer coefficients.

Write:

- $Q(x) = Q(x_1, x_r)$, x_1 first variable.
- $Q^1(x_1, x_r) = Q(x_1 + 1, x_r) - Q(x_1, x_r)$ (first difference)
  - ☛ smaller degree, still nonnegative integer coefficients.
  - ☛ Can compute $Q(x)$ with:

```
phi = Q(0,x_r);
for (i = 0; i < x; i++) {
  phi += Q1(i, x_r);
}
```

- Keep going until x_1 disappears.

```
phi = Q(0,x_r);
for (i = 0; i < x; i++) {
   // phi += Q1(i, x_r);
   phi += Q1(0, x_r);
   for (j = 0; j < i; j++) {
      phi += Q2(j, x_r);
   }
}
```

First ingredient (Feautrier): build multivariate polynomials

$Q(x_1, \ldots, x_n)$: multivariate polynomial, nonnegative integer coefficients.

Write:

- $Q(x) = Q(x_1, x_r)$, x_1 first variable.
- $Q^1(x_1, x_r) = Q(x_1 + 1, x_r) - Q(x_1, x_r)$ (first difference)
  - ☛ smaller degree, still nonnegative integer coefficients.
  - ☛ Can compute $Q(x)$ with:

```
phi = Q(0,x_r);
for (i = 0; i < x; i++) {
  phi += Q1(i, x_r);
}
```

- Keep going until x_1 disappears.

```
phi = Q(0,x_r);
for (i = 0; i < x; i++) {
   // phi += Q1(i, x_r);
   phi += Q1(0, x_r);
   for (j = 0; j < i; j++) {
      phi += Q2(j, x_r);
   }
}
```

- Continue with other variables:

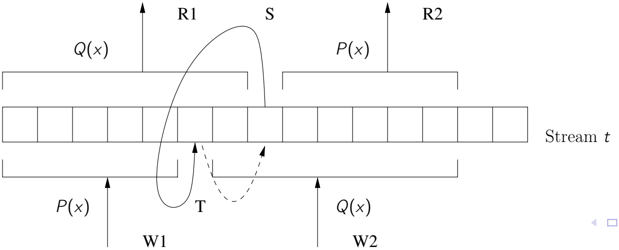
```
phi = Q(0,x_r); // Put new loops
for (i = 0; i < x; i++) {
   // phi += Q1(i, x_r);
   phi += Q1(0, x_r); // Put new loops
   for (j = 0; j < i; j++) {
      phi += Q2(j, x_r); // Put new loops
   }
}
```

Second ingredient: build the OpenStream structure

```
s, t streams;
for (x in D) {
  /* D is the n-dim. first orthant or
  the n-dim. cube of size N in it */
  R1: read Q(x) times in t;
  W1: write P(x) times in t;
  S: read once in t and write once in s;
  T: read once in s and write once in t;
  R2: read P(x) times in t;
  W2: writes Q(x) times in t;
}
```

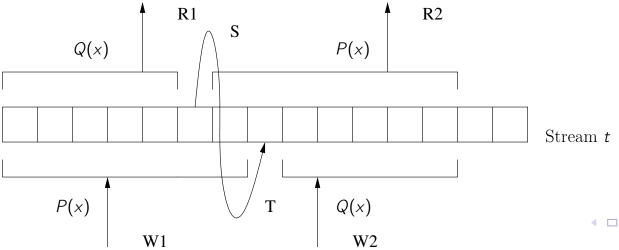
Deadlock situations:

- General case: iff $P(x) = Q(x)$.
- Kahnian case: iff $P(x) \leq Q(x)$.
  Note: iff no **causal** schedule.



Second ingredient: build the OpenStream structure

```
s, t streams;
for (x in D) {
  /* D is the n-dim. first orthant or
  the n-dim. cube of size N in it */
  R1: read Q(x) times in t;
  W1: write P(x) times in t;
  S: read once in t and write once in s;
  T: read once in s and write once in t;
  R2: read P(x) times in t;
  W2: writes Q(x) times in t;
}
```



Deadlock situations:

- General case: iff $P(x) = Q(x)$.
- Kahnian case: iff $P(x) \leq Q(x)$.
  Note: iff no **causal** schedule.

Second ingredient: build the OpenStream structure

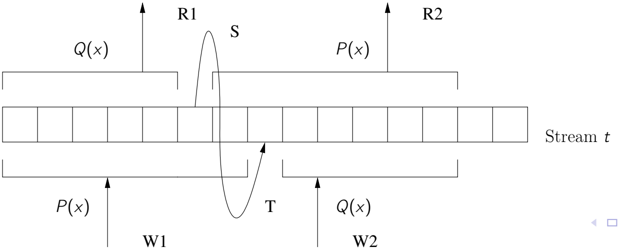
```
s, t streams;
for (x in D) {
  /* D is the n-dim. first orthant or
  the n-dim. cube of size N in it */
  R1: read Q(x) times in t;
  W1: write P(x) times in t;
  S: read once in t and write once in s;
  T: read once in s and write once in t;
  R2: read P(x) times in t;
  W2: writes Q(x) times in t;
}
```

Deadlock situations:

- General case: iff $P(x) = Q(x)$.
- Kahnian case: iff $P(x) \leq Q(x)$.
  Note: iff no **causal** schedule.

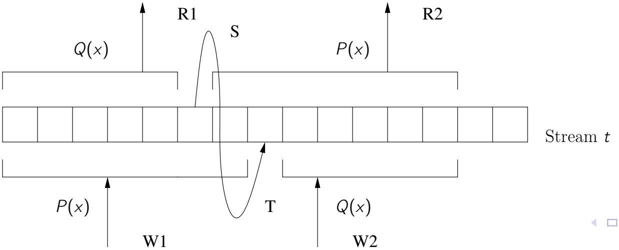
☛ 10th Hilbert's problem:

- $R(x) = 0$ iff $R^{+}(x) = R^{-}(x)$.
- $R(x) = 0$ iff $R^{2}(x) \leq 0$.



Second ingredient: build the OpenStream structure

```
s, t streams;
for (x in D) {
  /* D is the n-dim. first orthant or
  the n-dim. cube of size N in it */
  R1: read Q(x) times in t;
  W1: write P(x) times in t;
  S: read once in t and write once in s;
  T: read once in s and write once in t;
  R2: read P(x) times in t;
  W2: writes Q(x) times in t;
}
```



Deadlock situations:

- General case: iff $P(x) = Q(x)$.
- Kahnian case: iff $P(x) \leq Q(x)$.
  Note: iff no **causal** schedule.

☛ 10th Hilbert's problem:

- $R(x) = 0$ iff $R^+(x) = R^-(x)$.
- $R(x) = 0$ iff $R^2(x) \leq 0$.

Other problems:

- Missing producer.
- Bounded streams.

Take-home messages

About polyhedral specifications

- Polyhedral fragments to understand the limit of automation.
- Watch out: affine codes generate polynomials.
- Towards polynomial optimizations? In progress. See also Feautrier IMPACT'15.

Take-home messages

About polyhedral specifications

- Polyhedral fragments to understand the limit of automation.
- Watch out: affine codes generate polynomials.
- Towards polynomial optimizations? In progress. See also Feautrier IMPACT'15.

About OpenStream and Kahn Process Networks

- Interesting intermediate model: CSDF $<$ polyhedral OpenStream $<$ KPN.
- KPN: Turing-complete because model includes BDF (Buck/Parks).
- But BDF can react on values in streams (unlike polyhedral OpenStream).
- OpenStream with bounded buffers: not fully understood.
- Code optimizations (e.g., granularity change): not understood yet.

Take-home messages

About polyhedral specifications

- Polyhedral fragments to understand the limit of automation.
- Watch out: affine codes generate polynomials.
- Towards polynomial optimizations? In progress. See also Feautrier IMPACT'15.

About OpenStream and Kahn Process Networks

- Interesting intermediate model: CSDF < polyhedral OpenStream < KPN.
- KPN: Turing-complete because model includes BDF (Buck/Parks).
- But BDF can react on values in streams (unlike polyhedral OpenStream).
- OpenStream with bounded buffers: not fully understood.
- Code optimizations (e.g., granularity change): not understood yet.

About parallel languages and their analysis/optimization

- What do you prefer: deadlocks or races?
- How to express link between user/compiler and compiler/runtime?
- Parallel constructs can help dep. analysis (e.g., Chatarasi et al. IMPACT/PACT'15).

☛ Towards the analysis of parallel languages, with better user/compiler and compiler/runtime interactions (see also next talk on liveness analysis).