Polyèdres et compilation

François Irigoin & Mehdi Amini & Corinne Ancourt & Fabien Coelho & Béatrice Creusillet & Ronan Keryell

MINES ParisTech - Centre de Recherche en Informatique

12 May 2011
Our historical goals

- Find large grain data and task parallelism
  
  *includes medium and fine grain parallelism*

Still holding today for manycore and GPU computing!
Our historical goals

- Find large grain data and task parallelism
  *includes medium and fine grain parallelism*
- Interprocedural analyses: whole program compilation
  - *full inlining is ineffective because of complexity*
  - *cannot cope with recursion*

Still holding today for manycore and GPU computing!
Our historical goals

- Find large grain data and task parallelism
  *includes medium and fine grain parallelism*

- Interprocedural analyses: whole program compilation
  - *full inlining is ineffective because of complexity*
  - *cannot cope with recursion*

- Interprocedural transformations
  *selective inlining and outlining and cloning are useful*

Still holding today for manycore and GPU computing!
Our historical goals

- Find large grain data and task parallelism
  *includes medium and fine grain parallelism*

- Interprocedural analyses: whole program compilation
  - *full inlining is ineffective because of complexity*
  - *cannot cope with recursion*

- Interprocedural transformations
  *selective inlining and outlining and cloning are useful*

- No restrictions on input code...
  *Fortran 77, 90, C, C99*

Still holding today for manycore and GPU computing!
Our historical goals

- Find large grain data and task parallelism
  
  *includes medium and fine grain parallelism*

- Interprocedural analyses: whole program compilation
  
  - *full inlining is ineffective because of complexity*
  
  - *cannot cope with recursion*

- Interprocedural transformations
  
  *selective inlining and outlining and cloning are useful*

- No restrictions on input code...
  
  *Fortran 77, 90, C, C99*

- Hence decidability issues ⟷ over-approximations

Still holding today for manycore and GPU computing!
Our historical goals

- Find large grain data and task parallelism
  includes medium and fine grain parallelism
- Interprocedural analyses: whole program compilation
  - full inlining is ineffective because of complexity
  - cannot cope with recursion
- Interprocedural transformations
  selective inlining and outlining and cloning are useful
- No restrictions on input code...
  Fortran 77, 90, C, C99
- Hence decidability issues \(\implies\) over-approximations
- But exact analyses when possible

Still holding today for manycore and GPU computing!
Polyhedral School of Fontainebleau... vs Polytope Model

- Summarization/abstraction vs exact information
- No restrictions on input code
What can we do with polyhedra?

- Refine Bernstein’s conditions
What can we do with polyhedra?

- Refine Bernstein’s conditions
- Scheduling: loop parallelization, loop fusion
What can we do with polyhedra?

- Refine Bernstein’s conditions
- Scheduling: loop parallelization, loop fusion
- Memory allocation: privatization, statement isolation
What can we do with polyhedra?

- Refine Bernstein’s conditions
- Scheduling: loop parallelization, loop fusion
- Memory allocation: privatization, statement isolation
- And many more transformations:
  - Control simplification, constant propagation, partial evaluation, induction variable substitution, scalarization, inlining, outlining, invariant generation, property proof, memory footprint, dead code elimination...
What can we do with polyhedra?

- Refine Bernstein’s conditions
- Scheduling: loop parallelization, loop fusion
- Memory allocation: privatization, statement isolation
- And many more transformations: *Control simplification, constant propagation, partial evaluation, induction variable substitution, scalarization, inlining, outlining, invariant generation, property proof, memory footprint, dead code elimination*...
- Code synthesis
Simplified notations

- Identifiers $i, a$, Locations $l$ or $(a, \phi)$, Values
- Environment: $\rho : \text{Id} \rightarrow \text{Loc}$
- Memory state: $\sigma : \text{Loc} \rightarrow \text{Val}$, $\sigma(i) = 0$
- Preconditions: $P(\sigma) \subset \Sigma$
  - $\text{for}(i=0; \ i<n; \ i++) \{ \ldots \} \rightarrow \{ \sigma \mid 0 \leq \sigma(i) < \sigma(n) \}$
- Transformers: $T(\sigma, \sigma') \subset \Sigma \times \Sigma$
  - $i++; \rightarrow \{ (\sigma, \sigma') \mid \sigma'(i) = \sigma(i) + 1 \}$
- Array regions: $R : \Sigma \rightarrow \text{Loc}$
  - $a[i] \rightarrow \sigma \rightarrow \{ (a, \phi) \mid \phi = \sigma(i) \}$
- Statements $S, S_1, S_2 \in \Sigma \rightarrow \Sigma$
  - function call, sequence, test, loop, CFG...
- Programs $\Pi$: a statement
Convex Array Region for a Reference

\[ W_{S_1}(\sigma) = \{(a, \phi) | 0 \leq \phi < 5\} \]

for (i=0; i<5; i++)

\[ W_{S_2}(\sigma) = \{(a, \phi) | \phi = \sigma(i) [ \land 0 \leq \sigma(i) < 5 ] \} \]

a[i]=0.;
Convex Array Regions of Statement $S$

Property of a written region $W_S$

\[ \forall \sigma \quad \forall l \notin W_S(\sigma), \quad \sigma(l) = (S(\sigma))(l) \quad (1) \]

Note: The property holds for any over-approximation $\overline{W_S}$ of $W_S$.

Property of a read region $R_S$

\[ \forall \sigma \quad \forall \sigma' \]
\[ \forall l \in R_S(\sigma), \sigma(l) = \sigma'(l) \implies \begin{cases} R_S(\sigma) = R_S(\sigma') \\ W_S(\sigma) = W_S(\sigma') \\ \forall l \in W_S(\sigma), (S(\sigma))(l) = (S(\sigma'))(l) \end{cases} \quad (2) \]

Note: The property holds for any over-approximation $\overline{R_S}$ of $R_S$ in the left-hand side, but not for other over-approximations.
Conditions to exchange two statements $S_1$ and $S_2$

Evaluation of $S_1; S_2$: $\sigma \xrightarrow{S_1} \sigma_1 \xrightarrow{S_2} \sigma_{12}$

Assumptions:

\[
\forall \sigma, \quad W_{S_1}(\sigma) \cap R_{S_2}(\sigma_1) = \emptyset \quad (3)
\]

\[
\forall \sigma, \quad W_{S_1}(\sigma) \cap W_{S_2}(\sigma_1) = \emptyset \quad (4)
\]

Final state $\sigma_{12}$:

\[
(3) \quad \forall l \in R_{S_2}(\sigma_1), \quad l \notin W_{S_1}(\sigma) \xrightarrow{(1)} \sigma_1(l) = \sigma(l)
\]

\[
(2) \quad \left\{
\begin{array}{l}
R_{S_2}(\sigma_1) = R_{S_2}(\sigma) \\
W_{S_2}(\sigma_1) = W_{S_2}(\sigma) \\
\forall l \in W_{S_2}(\sigma), \quad \sigma_{12}(l) = \sigma_2(l)
\end{array}
\right.
\]

\[
(4) \quad \forall l \in W_{S_1}, \quad l \notin W_{S_2} \implies \sigma_{12}(l) = \sigma_1(l)
\]

\[
\forall l \notin W_{S_1} \cup W_{S_2}, \quad \sigma(l) = \sigma_1(l) = \sigma_{12}(l)
\]
Conditions to exchange two statements $S_1$ and $S_2$

Evaluation of $S_2; S_1$: $\sigma \xrightarrow{S_2} \sigma_2 \xrightarrow{S_1} \sigma_{21}$

Assumptions:

\[ \forall \sigma, \ W_{S_2}(\sigma) \cap R_{S_1}(\sigma_2) = \emptyset \] (5)

\[ \forall \sigma, \ W_{S_2}(\sigma) \cap W_{S_1}(\sigma_2) = \emptyset \] (6)

Final state $\sigma_{21}$:

(5) $\forall l \in R_{S_1}(\sigma_2), \ l \notin W_{S_2}(\sigma) \xrightarrow{(1)} \sigma_2(l) = \sigma(l)$

(2) $\rightarrow \left\{ \begin{array}{l} R_{S_1}(\sigma_2) = R_{S_1}(\sigma) \\ W_{S_1}(\sigma_2) = W_{S_1}(\sigma) \\ \forall l \in W_{S_1}(\sigma_2), \ \sigma_{21}(l) = \sigma_1(l) \end{array} \right.$

(6) $\forall l \in W_{S_2}(\sigma), \ l \notin W_{S_1}(\sigma_2) \Rightarrow \sigma_{21}(l) = \sigma_2(l)$

$\forall l \notin W_{S_1}(\sigma_2), \cup W_{S_2}(\sigma) \quad \sigma(l) = \sigma_2(l) = \sigma_{21}(l)$
Bernstein’s Conditions to Exchange $S_1$ and $S_2$

- Necessary condition:

\[ \forall \sigma \quad \text{let } \sigma_1 = S_1(\sigma), \sigma_2 = S_2(\sigma) \]

\[ \begin{align*}
WS_1(\sigma) \cap RS_2(\sigma_1) &= \emptyset \\
WS_1(\sigma) \cap WS_2(\sigma_1) &= \emptyset \\
WS_2(\sigma) \cap RS_1(\sigma_2) &= \emptyset \\
WS_2(\sigma) \cap WS_1(\sigma_2) &= \emptyset
\end{align*} \]

\[ \implies \begin{align*}
WS_1(\sigma) \cap RS_2(\sigma) &= \emptyset \\
WS_1(\sigma) \cap WS_2(\sigma) &= \emptyset \\
WS_2(\sigma) \cap RS_1(\sigma) &= \emptyset \\
WS_2(\sigma) \cap WS_1(\sigma) &= \emptyset
\end{align*} \]

according to the two previous slides.
Starting from Bernstein’s conditions

- Let’s assume: \( \forall \sigma \ W_{S_1}(\sigma) \cap R_{S_2}(\sigma) = \emptyset \)
- This implies by (1): \( \forall \sigma \ \forall l \in R_{S_2} \ (S_1(\sigma))(l) = \sigma(l) \)
- Hence by (2): \( \forall \sigma \ R_{S_2}(\sigma_1) = R_{S_2}(\sigma) \land W_{S_2}(\sigma_1) = W_{S_2}(\sigma) \)

- In the same way: \( \forall \sigma \ W_{S_2}(\sigma) \cap R_{S_1}(\sigma) = \emptyset \)
- Implies: \( R_{S_1}(\sigma_2) = R_{S_1}(\sigma) \land W_{S_1}(\sigma_2) = W_{S_1}(\sigma) \)

So Bernstein’s conditions are sufficient to prove:

\[ \forall \sigma \ (S_1; S_2)(\sigma) = (S_2; S_1)(\sigma) \]
Coarse grain parallelization of a loop

- Let’s assume convex array regions $R_B$ and $W_B$ for the loop body.
- Let $P_B$ be the body precondition and $T_{B,B}^+$ the inter-iteration transformer.
- Direct parallelization of a loop using convex array regions with Bernstein’s conditions for the iterations of the body $B$:
  - $\forall v \in \text{Id} \quad \forall \sigma, \sigma' \in P_B \text{ s.t. } T_{B,B}^+(\sigma, \sigma')$
  - $R_{B,v}(\sigma) \cap W_{B,v}(\sigma') = \emptyset$
  - $R_{B,v}(\sigma') \cap W_{B,v}(\sigma) = \emptyset$
  - $W_{B,v}(\sigma) \cap W_{B,v}(\sigma') = \emptyset$
- Note: $T_{B,B}^+(\sigma, \sigma') \Rightarrow \sigma(i) < \sigma'(i)$ where $i$ is the loop index.
- Each iteration can be interchanged with any other one.
- No dependence graph, no restrictions on loop body, no restriction on control, no restriction on references, no restriction on loop bounds...
Coarse Grain Parallelization of a Loop with Privatization

- Beyond Bernstein’s conditions, use $IN_B$ and $OUT_B$ array regions instead of $R_B$ and $W_B$ regions.
- Insure non-interference for interleaved execution: privatization or expansion for locations in $W_B - OUT_b$.
- $OUT_B$ can be over-approximated with $\overline{OUT_B}$ because it is used to decide the parallelization.
- $W_B$ cannot be overapproximated.
- Must be combined with reduction detection.
Fusion of Loops $L_1$ and $L_2$ with delay $d$

- for(i1...) S1; for(i2...) S2
- initial schedule:

  \[
  S_1^0 \rightarrow S_1^1 \rightarrow S_1^2 \rightarrow S_1^3 \rightarrow S_1^4
  \]

- new schedule:

  \[
  S_1^0 \rightarrow S_1^1 \rightarrow S_2^0 \rightarrow S_2^1 \rightarrow S_2^2 \rightarrow S_2^3 \rightarrow S_2^4 \rightarrow
  \]

- for(...) S1; for(...) [S1;S2] for(...) S2
Fusion of Loops $L_1$ and $L_2$ with delay $d$: Legality

- Assumes convex array regions $R_1$ and $W_1$ for body $B_1$ of loop $L_1$ with index $i_1$, $R_2$ and $W_2$ for body $B_2$ of loop $L_2$ with index $i_2$

- Permutation of the last iterations of $L_1$ and the first iterations of $L_2$ with a delay $d$:

$$\forall \sigma_1 \forall \sigma_2 \ P_1(\sigma_1) \land P_2(\sigma_2) \land T_{12}(\sigma_1, \sigma_2) \land \sigma_1(i_1) > \sigma_2(i_2) + d$$

$$R_1(\sigma_1) \cap W_2(\sigma_2) = \emptyset$$

$$R_2(\sigma_2) \cap W_1(\sigma_1) = \emptyset$$

$$W_1(\sigma_1) \cap W_2(\sigma_2) = \emptyset$$

- $P_1$, $P_2$, $T_{12}$, $R_1$, $W_1$, $R_2$, $W_2$ can be all over-approximated

- Check emptiness of convex sets for a polyhedral instantiation

- No restrictions on $B_1$ nor $B_2$ nor the loop index identifiers or ranges
Fusion of Loops $L_1$ and $L_2$ with delay $d$: Profitability

- Reduce memory loads:
  \[
  \left( \bigcup_{\sigma_1 \in P_1} R_1(\sigma_1) \right) \cap \bigcup_{\sigma_2 \in P_2 \cap T_{1,2}(\sigma_1)} R_2(\sigma_2) \neq \emptyset
  \]

- Avoid intermediate store and reloads:
  \[
  \left( \bigcup_{\sigma_1 \in P_1} W_1(\sigma_1) \right) \cap \bigcup_{\sigma_2 \in P_2 \cap T_{1,2}(\sigma_1)} R_2(\sigma_2) \neq \emptyset
  \]

- With minimal cache size:
  \[
  \left( \bigcup_{\sigma_1 \in P_1} (R_1(\sigma_1) \cup W_1(\sigma_1)) \right) \cup \bigcup_{\sigma_2 \in P_2 \cap T_{1,2}(\sigma_1)} (R_2(\sigma_2) \cup W_2(\sigma_2))
  \]
Array privatization

- An array $a$ is privatizable in a loop $l$ with body $B$ if

$$\forall \sigma \in P_B, \quad IN_{B,a}(\sigma) = OUT_{B,a}(\sigma) = \emptyset$$

- $IN_{B,a}$ is the set of elements of $a$ whose input values are used in $B$. For a sequence $S_1; S_2$:

$$IN_{S_1;S_2} = IN_{S_1} \cup \left((IN_{S_2} \circ T_{S_1}) - W_{S_1}\right)$$

- $OUT_{B,a}(\sigma)$ is the set of elements of $a$ whose output values are used by the continuation of $B$ executed in memory state $\sigma$. For a sequence $S_1; S_2$:

$$OUT_{S_1} = (OUT_{S_1;S_2} - W_{S_2} \circ T_{S_1}) \cup (W_{S_1} \cap IN_{S_2} \circ T_{S_1})$$
Examples of IN and OUT regions

- Source code for function `foo`
  ```c
  void foo(int n, int i, int a[n], int b[n]) {
    a[i] = a[i]+1;
    i++;
    b[i] = a[i]; }
  ```

- Source code for `main`:
  ```c
  int main() {
    int a[100], b[100], i;
    foo(100, i, a, b);
    printf("%d\n", b[0]); }
  ```

- R, W, IN and OUT array regions for call site to `foo`:
  ```c
  // <a[PHI1]-R-EXACT-{i<=PHI1, PHI1<=i+1}>
  // <a[PHI1]-W-EXACT-{PHI1==i}>
  // <b[PHI1]-W-EXACT-{PHI1==i+1}>
  
  // <a[PHI1]-IN-EXACT-{i<=PHI1, PHI1<=i+1}>
  
  // <b[PHI1]-OUT-EXACT-{PHI1==0, PHI1==i+1}>
  ```

  ```c
  foo(100, i, a, b);
  ```
Properties of IN regions

- If two states $\sigma$ and $\sigma'$ assign the same values to the locations in $IN_S$, statement $S$ produces the same trace with $\sigma$ and $\sigma'$:

$$\forall \sigma \quad \forall \sigma' \quad (7)$$

$$\forall l \in IN_S(\sigma), \sigma(l) = \sigma'(l) \Rightarrow$$

$$\begin{cases} 
R_S(\sigma) = R_S(\sigma') \\
W_S(\sigma) = W_S(\sigma') \\
IN_S(\sigma) = IN_S(\sigma') \\
\forall l \in W_S(\sigma), (S(\sigma))(l) = (S(\sigma'))(l)
\end{cases}$$

- Almost identical to property for $R$ regions
- But also $\forall \sigma \quad \forall \sigma'$:

$$\forall l \notin \bigcup_{\sigma \in P_S} \left( R_S(\sigma) \setminus IN_S(\sigma) \right), \sigma(l) = \sigma'(l) \Rightarrow Equivalent_s(\sigma, \sigma')$$
Properties of OUT regions

- The values of variables written by $S$ but not used later do not matter:

$$\forall \sigma, \forall \sigma', \forall l \notin \bigcup_{\sigma \in P_S} \left( W_S(\sigma) - OUT_S(\sigma) \right),$$

$$(S(\sigma))(l) = (S(\sigma'))(l) \Rightarrow Equivalent_C(\sigma, \sigma')$$

- In other words, statement $S$ can be substituted by statement $S'$ in Program $\Pi$ if they only differ by writing different values in memory locations that are not read by the continuation.
Scalarization

- Replace a set of array references by references to a local scalar:
  \[
  a[j]=0; \; \text{for}(i...) \; \{ \; \ldots \; a[j]=a[j]*b[i];\; \ldots \; \} \\
  \rightarrow s=0; \; \text{for}(i...) \; \{ \; \ldots \; s*=b[i];\; \ldots \; \} \; a[j]=s;
  \]

- Let \( B \) and \( i \) be a loop body and index, and \( W_B \) the write region function.

- Sufficient condition: each loop iteration accesses only one array element.

- Let
  \[
  f : Val \rightarrow \mathcal{P}(\Phi) \; \text{s.t.} \; f(v) = \{ \phi \mid \exists \sigma : \sigma(i) = v \land (a, \phi) \in W_B(\sigma) \}
  \]

- If \( f \) is a mapping \( Val \rightarrow \Phi \), array \( a \) can be replaced by a scalar.

- Initialization and exportation according to \( IN_B \) and \( OUT_B \).
Statement Isolation

- Goal: replace \( S \) by a new statement \( S' \) executable with a different memory \( M' \):
  
  \[
  i = i + 1; \\
  \rightarrow \{ \text{int } j; \ j = i; \ j = j + 1; \ i = j; \}
  \]

- Let \( S \) be a statement with regions \( R_S, W_S, IN_S \) and \( OUT_S \).
- Declare new variables \( new(l) \) for \( l \in \bigcup_{\sigma \in P_S} (R_S(\sigma) \cup W_S(\sigma)) \)
- Copy in: \( \forall l \in IN_S(\sigma) \ M'[new(l)] = M[l] \)
- Substitute all references to \( l \) by references to \( new(l) \) in \( S \)
- Copy out: \( \forall l \in OUT_S(\sigma) \ M[l] = M'[new(l)] \)
Statement Isolation

• Goal: replace $S$ by a new statement $S'$ executable with a different memory $M'$:
  
  \[
  i = i + 1; \\
  \rightarrow \{ \text{int } j; \ j = i; \ j = j + 1; \ i = j; \}
  \]

• Let $S$ be a statement with regions $R_S$, $W_S$, $IN_S$ and $OUT_S$.

• Declare new variables $new(l)$ for $l \in \bigcup_{\sigma \in P_S} (R_S(\sigma) \cup W_S(\sigma))$

• Copy in: $\forall l \in IN_S(\sigma) \ M'[new(l)] = M[l]$

• Substitute all references to $l$ by references to $new(l)$ in $S$

• Copy out: $\forall l \in OUT_S(\sigma) \ M[l] = M'[new(l)]$

• Copy out fails because of over-approximations of $OUT_S$!

• Copy in: $\forall l \in (IN_S(\sigma)) \cup (OUT_S(\sigma)) \ M'[new(l)] = M[l]$

• related to outlining and privatization and localization
**Induction variable substitution**

- Substitute \( k \) by its value, function of the loop index \( i \):

  \[
  k=0; \quad \text{for}(i=0;...) \quad \{ k+=2; \ b[k] = ... \}
  \]

  \[
  \text{for}(i=0;...) \quad \{ \ b[2*i+2] = ... \}
  \]

- Variable \( k \) can be substituted in statement \( S \) with precondition \( P_S \) within a loop of index \( i \) if \( P_S \) defines a mapping from \( \sigma(i) \) to \( \sigma(k) \):

  \[
  \nu \rightarrow \{ \nu' | \exists \sigma \in P_S \ \sigma(i) = \nu \land \sigma(k) = \nu' \}
  \]
Constant Propagation

- Replace references by constants:
  \[
  \text{if}(j==3) \ a[2*j+1]=0; \\
  \text{if}(j==3) \ a[7]=0;
  \]

- An expression \( e \) can be substituted under precondition \( P \) if:
  \[
  \left| \{ \nu \in Val \mid \exists \sigma \in P \ \nu = E(e, \sigma) \} \right| = 1
  \]

- Simplify expressions:
  \[
  \text{if}(i+j==n) \ a[i+j]=0; \\
  \text{if}(i+j==n) \ a[n]=0;
  \]
If you insist on:
- using an algorithm with restricted applicability
- reducing locality with loop distribution

Use array regions to deal at least with procedure calls

Dependence system for two regions of array \( a \) in statements \( S_1 \) and \( S_2 \) in a loop nest \( \vec{i} \):

\[
\{(\sigma_1, \sigma_2) \mid \sigma_1(\vec{i}) \prec \sigma_2(\vec{i}) \land T_{S_1,S_2}(\sigma_1, \sigma_2) \land P_{S_1}(\sigma_1) \land P_{S_2}(\sigma_2) \land R^{a}_{S_1}(\sigma_1) \cap W^{a}_{S_2}(\sigma_2) \} = \emptyset
\]  

(9)

Useful for tiling, which includes all unimodular loop transformations
Dead code elimination

- Remove unused definitions:

  ```c
  int foo(int i) { int j = i + 1; i = 2; int k = i + 1; return j; }
  ```

  →

  ```c
  int foo(int i) { int j = i + 1; return j; }
  ```

- Useless? See some automatically generated code
- Useless? See some manually maintained code 😊
- Any statement S with no OUT\_S region?
- Possible, but not efficient with the current semantics of OUT regions in PIPS
Time-out!

- Declarations
- Control
- Communications
- Copy operations
Conclusion: simple polyhedral conditions in a compiler

- Difficulties hidden in a few analyses, available with PIPS: \( T, P, W, R, \text{IN}, \text{OUT} \)
- Legality of many program transformations can be checked with analyses: mapping, function, empty set, ...
- Yes, quite often: Control simplification, constant propagation, partial evaluation, induction variable substitution, privatization, scalarization, coarse grain loop parallelization, loop fusion, statement isolation, ...
- But not always: graph algorithms are useful too Dead code elimination, ... with OUT regions?
Conclusion: what might go wrong with polyhedra?

- The analysis you need is not available in PIPS: re-use existing analyses to implement it.
- Its accuracy is not sufficient: implement a dynamic analysis (a.k.a. instrumentation).
- The worst case complexity is exponential: exceptions are necessary for recovery.
- Monotonicity of results on space, time or magnitude exceptions: more work needed, exploit parallelism within PIPS.
- Possible recomputation of analyses after each transformation: more work needed, composite transformations...
Conclusion: see what is available in PIPS!

- Many more program transformations
- Pointer analyses are improving
- Try PIPS with no installation cost: http://paws.pips4u.org
  
  *On-going work... Do not overload our PIPS server 😊*

- Or install it: http://pips4u.org

- Or install Par4all: http://www.par4all.org

- Or simply talk to us!
Questions?