Hybrid Analysis
and its Application to Automatic Parallelization

Silvius Rus
Lawrence Rauchwerger
Thread Level Loop Parallelization

Sequential Loop

\[
\begin{align*}
\text{DO } & j = 1, 100 \\
a(j) = 0 \\
\text{ENDDO}
\end{align*}
\]

OpenMP Directives

\[
\begin{align*}
\$OMP \text{ PARALLEL DO}
\text{DO } & j = 1, 100 \\
a(j) = 0 \\
\text{ENDDO}
\end{align*}
\]

Multithreaded Code

\[
\begin{align*}
\text{SHARED } & a \\
\text{PRIVATE } & \text{begin, end} \\
\text{begin} = & 10*\text{thread}_\text{id}+1 \\
\text{end} = & \text{begin}+9 \\
\text{DO } & j = \text{begin}, \text{end} \\
a(j) = 0 \\
\text{ENDDO}
\end{align*}
\]

(assuming 10 threads)

Dependence Analysis

Privatization (renaming)

Reduction parallelization

...
Data Dependence (DD):

- Data dependence relations are used as the essential ordering constraints among statements or operations in a program.
- Data dependence happens when two operations access the same memory location and at least one of them writes to the location.

Three basic data dependence relations:

- **Flow**: \[ x = \ldots \]
  \[ \ldots = x \]
- **Anti**: \[ x = \ldots \]
  \[ x = \ldots \]
- **Output**: \[ \ldots = x \]
  \[ x = \ldots \]
Can a loop be executed in parallel?

- Test procedure
  - FOR every pair of load/store and store/store operations: \(<L,S>\) DO
  - IF (L and S could access the same location in diff. iterations)
  - LOOP is sequential

- For arrays, memory accesses are functions of loop indices. These functions can be: linear, non-linear, unknown map

A parallel loop

\[
\text{DO } i = \ldots \\
\]

A sequential loop

\[
\text{DO } i = \ldots \\
\]
Static Data Dependence Analysis

Dependence set = solutions to a system of linear inequations
Independence  $\iff$ no integer solutions
GCD, Banerjee, Range Test, Omega

```
DO  j=1,10
    a(j)=a(j+40)
ENDDO
```

No cross iteration dependences $\iff$ No integer solutions:

\[
\begin{align*}
1 \leq j_1 \leq 10 \\
1 \leq j_2 \leq 10 \\
j_1 \neq j_2 \\
j_1 = j_2 + 40
\end{align*}
\]
Input-Sensitive Decisions

```
READ *, N
DO  j=1,N
  a(j)=a(j+40)
ENDDO
```

WRITE \hspace{1cm} READ

\begin{align*}
N = 5: & \quad [1: 5] \cap [41:45] = \emptyset \quad \text{Independent} \\
N = 45: & \quad [1:45] \cap [41:85] = [41:45] \quad \text{Dependent}
\end{align*}

Different outcomes depending on the value of $N$.

*Compile-time analysis fails!*
Run-time Analysis

LRPD Test

1. Instrumentation of all relevant memory references

2. Run-time analysis of the resulting trace

N = 5

Predicate Extraction

Extract conditions under which there are no dependencies

Only simple cases!
A Motivating Example

Is a independent in the outermost loop?

Assume at run-time:

offsets(j) = (j-1)*N^2

- Compile-time analysis: NO
- Run-time analysis:
  - Reference by reference: YES, cost = O(N^3)
- Predicate extraction: NO
- Ideal: YES, cost = O(N) (actual needed work)
## Compile-time vs. Run-time

<table>
<thead>
<tr>
<th><strong>Compile Time</strong></th>
<th><strong>Run-time, reference by reference</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PROs</strong></td>
<td><strong>PROs</strong></td>
</tr>
<tr>
<td>– No run-time overhead</td>
<td>– Always finds answers</td>
</tr>
<tr>
<td><strong>CONs: too conservative</strong></td>
<td><strong>CONs</strong></td>
</tr>
<tr>
<td>– Dependence on input values</td>
<td>– Run-time overhead proportional to dynamic memory reference count</td>
</tr>
<tr>
<td>– Weak symbolic analysis</td>
<td>– Ignores partial compile-time analysis results</td>
</tr>
<tr>
<td>- Subscripted subscripts</td>
<td></td>
</tr>
<tr>
<td>- Complex recurrences</td>
<td></td>
</tr>
<tr>
<td>- Address-data computation cycles</td>
<td></td>
</tr>
<tr>
<td>– Impractical symbolic analysis</td>
<td></td>
</tr>
<tr>
<td>- Combinatorial explosion</td>
<td></td>
</tr>
</tbody>
</table>
## Hybrid Analysis of Memory Reference Patterns

<table>
<thead>
<tr>
<th></th>
<th>Compile-time Analysis</th>
<th>Hybrid Analysis</th>
<th>Run-time Analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>STATIC</strong></td>
<td>Symbolic analysis</td>
<td>Symbolic analysis</td>
<td></td>
</tr>
<tr>
<td><strong>DYNAMIC</strong></td>
<td></td>
<td>Continue analysis with actual values</td>
<td>Full reference-by-reference analysis</td>
</tr>
</tbody>
</table>

**Framework**: Memory reference pattern analysis

**Application**: Automatic parallelization
READ *, N
DO j=1,N
   a(j)=a(j+40)
ENDDO

Are there any cross-iteration dependences?

1. Collect references.

2. Aggregate them symbolically.

3. Formulate independence test.

4. Extract lowest-cost runtime test.
Aggregation Across an Iteration Space

- WRITE pattern for $a$:

  ```
  SUBROUTINE Rad(a, b)
  INTEGER a(*), b(*)
  DO j=1,100
      a(j) = 2*b(j) + 1
  ENDDO
  ```

  \([1:100]\)

- This case solved at compile-time: LMAD
Aggregation Into an Actual Context

SUBROUTINE Rad(a, b)
INTEGER a(*), b(*)
DO j=1,100
   a(j)=2*b(j)+1
ENDDO

- WRITE pattern for cc

   CALL Rad(cc, ch)

- This case also solved at compile-time
Gate Operator:
Postpone Analysis Failure due to an Unknown Predicate

- WRITE descriptor on `cc`:
- Cannot be solved at compile-time
  - `(na.EQ.0)` is not known

```fortran
SUBROUTINE Rfft(cc, ch)
  INTEGER na, cc(*), ch(*)
  na=1
  DO j=1,3
    na=1-na
    IF (na.EQ.0) THEN
      CALL Rad(cc, ch)
    ELSE
      CALL Rad(ch, cc)
    ENDIF
  ENDDO
END
```
Recurrence Operator
Postpone Analysis Failure due to a Recurrence with no Closed Form

- WRITE descriptor on cc
- Recurrence on na with no close form

```fortran
SUBROUTINE REEc(cc, ch)
INTEGER na, cc(*), ch(*)
na=1
DO j=1,3
na=1-na
IF (na.EQ.0) THEN
    CALL Rad(cc, ch)
ELSE
    CALL Rad(ch, cc)
ENDIF
ENDDO
```

[1:100] (na.EQ.0)
Translation Operator
Postpone Analysis Failure due to Translation Issues (reshaping etc)

*WF* pattern on array *w*

*na* - local to *Rfft*
Uniform Set of References (USR)

\[ T = \{ \text{LMAD, } \cap, \cup, -, (), \#, x, \Theta, \text{Gate, Recurrence, Call Site} \} \]

\[ N = \{ \text{USR} \} \]

\[ S = \text{USR} \]

\[ P = \{ \text{USR} \rightarrow \text{LMAD} | (\text{USR}) \]
\[ \text{USR} \rightarrow \text{USR} \cap \text{USR} \]
\[ \text{USR} \rightarrow \text{USR} \cup \text{USR} \]
\[ \text{USR} \rightarrow \text{USR} - \text{USR} \]
\[ \text{USR} \rightarrow \text{USR} \# \text{Gate} \]
\[ \text{USR} \rightarrow \text{USR} \times \text{Recurrence} \]
\[ \text{USR} \rightarrow \text{USR} \Theta \text{Call Site} \} \]

\[ \text{LMAD} = \text{Start} + [\text{Stride}_1:\text{Span}_1, \text{Stride}_2:\text{Span}_2, ...] \]
Reducing Complexity

- At compile-time
  - Contiguous aggregation, interleaving of LMADs
  - Loop invariant USR hoisting
  - Logic inference, e.g. $G\#D_1 \cap \overline{G}\#D_2 = \emptyset$
  - Set identities, e.g. $(A - B) - A = \emptyset$
  - Lattice properties, e.g. $A - T = \emptyset$

- At run-time
  - Contiguous aggregation, interleaving
Reference-by-reference pure RT test

Hybrid Analysis

Number of iterations necessary for computing access pattern

\[
100 \times 1000 \times 3 \times 100 = 30,000,000
\]
The USR’s Contribution: Static Analysis Failure Tolerance

- **USR** = Closed form representation
  - With respect to analysis operations
    - Set operations: ∪, ∩, -
    - Predication #, loop expansion ⊗, translation Θ
  - Over any structured program block
    - Loop body, If block, subroutine
    - Large, interprocedural blocks

- Previous representations
  - Are limited to
    - Linear subscripts, control predicates, loop bounds
    - Direct indexing only (no subscripted subscripts)
  - At points of failure
    - Stop analysis or
    - Approximate (often overly) conservatively
Memory Classification Analysis

- Memory Classification Analysis
  - RO: only read
  - WF: written before any read
  - RW: all other cases

- Aggregate information across program
  - RO, WF, RW as USRs
Are there any cross-iteration dependences?

1. Collect references.

2. Aggregate them symbolically.

3. Formulate independence test.

4. Extract lowest-cost runtime test.
Data Dependences

- Given:
  - Loop expression: $j = 1,N$
  - Per-iteration aggregated descriptors $RO_j, WF_j, RW_j$

- Solve equation $RO \cap WF = \phi$

- At compile-time:
  - $RO \cap WF$ evaluates to $\phi \Rightarrow$ independent
  - $RO \cap WF$ evaluates to a set that is definitely not empty $\Rightarrow$ dependent
  - All other cases: run-time dependence test
```fortran
READ *, N
DO j=1,N
   a(j)=a(j+40)
ENDDO
```

Are there any cross-iteration dependences?

1. Collect references.
2.Aggregate them symbolically.
3. Formulate independence test.
4. Extract lowest-cost runtime test.

\[ [41:40+N] \cap [1:N] \]

\( \Phi \)
Example: Exposed Read

**USR Equation**

```
DO i=1, 10
  DO j=1, 10
    IF (C(j,i).GT.0) THEN
      WORK(j) = ...
    ENDIF
  ENDDO
ENDDO
```

**Predicate Tree**

```
10 \land_{j=1}
C(j,i).GT.0
```
Proof System

Input: \textit{USR equation} $D = \phi$

Output: \textit{Predicate} $P$

Such that:

$P \iff D = \phi$

\textbf{OPTIMISTIC: Sufficient predicate:} \hspace{1cm} P \Rightarrow D = \phi$

\textbf{PESSIMISTIC: Necessary predicate:} \hspace{1cm} D = \phi \Rightarrow P, \text{ or } \overline{P} \Rightarrow D \neq \phi$
How Hard Is This Problem?

$A(1) = \ldots$

$j = f(x)$

$A(j) = \ldots$

What is $x$ such that $f(x) = 1$?

$P(f(x)) = \text{true} \iff x \in f^{-1}(P^{-1}(\text{true}))$

Worst case: $O(2^{\text{size}(x) \times \text{complexity}(P \cdot f)})$

But most real cases are tractable!
Our Solution:
Recursive Descent on the USR Tree

From Union To Conjunction

.FALSE.

.FALSE.

.FALSE.

.FALSE.

.FALSE.
Predicate Formal Specification

\[ T = \{ \text{Logical Expression, } \land, \lor, \otimes_\land, \otimes_\lor, \Theta, \text{ Recurrence, Call Site, Library Routine} \} \]

\[ N = \{ \text{PDAG} \} \]

\[ S = \text{PDAG} \]

\[ P = \{ \text{PDAG }\rightarrow\text{ Logical Expression} \]
\[
\text{PDAG }\rightarrow\text{ PDAG }\land\text{ PDAG} \\
\text{PDAG }\rightarrow\text{ PDAG }\lor\text{ PDAG} \\
\text{PDAG }\rightarrow\text{ PDAG }\otimes_\land\text{ Recurrence} \\
\text{PDAG }\rightarrow\text{ PDAG }\otimes_\lor\text{ Recurrence} \\
\text{PDAG }\rightarrow\text{ PDAG }\Theta\text{ Call Site} \\
\text{PDAG }\rightarrow\text{ Library Routine} \} \]
Grammar-directed Translation: Algorithm *Solve*

- **Input**: \( D=\emptyset \) (AST on USR grammar)
- **Output**: \( P \) (AST on PDAG grammar)

**CASE** \( \text{root}(D) \) **OF**

- **Leaf**: \( P = false \)
- **Union** \((A,B)\): \( P = \text{Solve}(A=\emptyset) \land \text{Solve}(B=\emptyset) \)
- **Intersection** \((A,B)\): \( P = \text{Solve}(A=\emptyset) \lor \text{Solve}(B=\emptyset) \lor \text{Solve Disjoint}(A, B) \)
- **Difference** \((A,B)\): \( P = \text{Solve}(A=\emptyset) \lor \text{Solve Inclusion}(A, B) \)
- **Predicate** \((p,A)\): \( P = \overline{p} \lor \text{Solve}(A=\emptyset) \)
- **Loop Expansion** \((A_i, i=1,N)\): \( P = \prod_{i=1,N} \text{Solve}(A_i) \)
- **Translation** \((\text{CallSite}, A)\): \( P = \text{Translate}(\text{CallSite}, \text{Solve}(A=\emptyset)) \)

**IF** \((P \text{ not equivalent to } (D=\emptyset))\) **Then** \( P = P \lor \text{Reference Based Test}(D=\emptyset) \)
Inclusion Example: Exposed Read

USR Equation

\[
\text{READ} = \{1:10\}
\]

\[
\text{WRITE} = \{x\}
\]

\[
\phi \subseteq \{x\} \subseteq \{C(j,i) \geq 0\} \subseteq \{1:10\}
\]

\[
\text{DO } i = 1, 10
\]
\[
\text{DO } j = 1, 10
\]
\[
\text{IF } (C(j,i) \geq 0) \text{ THEN}
\]
\[
\text{WORK}(j) = \ldots\]
\[
\text{ENDIF}
\]
\[
\text{ENDDO}
\]
\[
\text{ENDDO}
\]

\[
\ldots = \text{WORK}(1:10)
\]
Inclusion Example

Since READ is covered by the maximal WRITE, the condition is sufficient:

```
DO i=1, 10
  DO j=1,10
    IF (C(j,i).GT.0) THEN
      WORK(j) = ...
    ENDIF
  ENDDO
ENDDO
...
= WORK(1:10)
```

Extract predicate $p^{WRITE}$ under which WRITE is maximal.

$$p^{WRITE} = \bigwedge_{j=1}^{10} C(j,i).GT.0$$
Algorithm **Solve Inclusion**

- **Input**: A,D (AST on USR grammar)
- **Output**: P (AST on PDAG grammar)

**CASE** root(D) **OF**

- **Union(B,C)**: \( P = \text{Solve}(A-B=\phi) \lor \text{Solve}(A-C=\phi) \lor ? \) (see ⚫)
- **Intersection(B,C)**: \( P = \text{Solve}(A-B=\phi) \land \text{Solve}(A-C=\phi) \)
- **Difference(B,C)**: \( P = \text{Solve}(A-B=\phi) \lor \text{Solve}(A \cap C) \)
- **Predicate(q,B)**: \( P = q \land \text{Solve}(A-B=\phi) \)

**CASE** root(A) **OF**

... (more cases handled based on set algebra identities)

⚫ **IF** (\( P \) not equivalent to \( A \subseteq D \)) **THEN**

- Extract minimal (closest) predicated overestimate of A, \( \langle q_A, \lceil A \rceil \rangle \)
- Extract maximal (closest) predicated underestimate of D, \( \langle q_D, \lfloor D \rfloor \rangle \)
- \( P = q_A \land q_D \land \text{Solve Inclusion LMADs}(\lceil A \rceil, \lfloor D \rfloor) \)

**Similar for Solve Disjoint**
Fallback

Not all equations can be reversed

Solutions

Pattern library

Monotonic  Injectable

... Extensible Compiler!

Reference-based test

LRPD  USR
Monotonicity

\[ \phi \cap X \cap \cap \phi \cap \phi \phi \phi \]

\[ x \]

\[ x \]

\[ i=1,N \]

\[ \text{READ } *, N, \text{offsets}(1:N) \]
\[ \text{DO } i = 1, N \]
\[ \text{ind } = \text{offsets}(i) \]
\[ \text{DO } j = 1, N \]
\[ \text{DO } k = 1, N \]
\[ \text{ind } = \text{ind}+1 \]
\[ a(\text{ind}) = ... \]
\[ \text{ENDDO} \]
\[ \text{ENDDO} \]
\[ \text{ENDDO} \]

\[ \text{offsets}(i) + N^2 + N^2 \]

\[ \text{offsets}(h) + 1: \]
\[ \text{offsets}(h) + N^2 \]

\[ \text{offsets}(i)+1: \]
\[ \text{offsets}(i)+N^2 \]

\[ h=1,i-1 \]

\[ \text{offsets}(h)+1: \]
\[ \text{offsets}(h)+N^2 \]

\[ \text{offsets}(i)+N^2 \]
\[ \text{offsets}(i+1)+1 \]

\[ \text{ENDDO} \]
\[ \text{ENDDO} \]
\[ \text{ENDDO} \]

\[ O(N^3) \Rightarrow O(N) \]

Sufficient, but not necessary
Injectivity

\( \phi \)

\( X \)

\( i=1, N \)

offsets(i)+1:
offsets(i)+N*N

offsets(h)+1:
offsets(h)+N*N

h=1, i-1

1. Sort( offsets(1:N) )

2.

\( N-1 \)
\( \land \)
\( i=1 \)

offsets(i)+N*N
.offsets(i+1)+1

\( O(N^3) \Rightarrow O(N \log N) \)

Necessary and sufficient!

Also: \( O(N) \) sufficient test as monotonicity check
Extensible Compiler

- Pattern library
  - Memory reference patterns as USRs
  - USR → GXL/XML

- Pattern recognition
  - USR equivalence rules

- Canned solutions
  - Data dependence: monotonicity, sorting
  - Other uses: specific patterns
Reference-based Runtime Tests

Aggregated USR Evaluation

\[ \phi \cap [41:40+N] \cap [1:N] \]

READ \;
\;
WRITE

CALL build_USR(41, 40+N, D_0)
CALL build_USR(1, N, D_1)
CALL intersect(D_0, D_1, D_2)
isIndependent = check_empty(D_2)

USR → FORTRAN: attribute grammar

PRO: May reduce asymptotic complexity
CON: Uses expensive operations

Reference by reference
LRPD

DIMENSION a(100)
DIMENSION sa(100)
...
DO j=1,N
   CALL mark_write(sa, j)
   CALL mark_read(sa, j+40)
ENDDO
isIndependent =
is_disjoint_read_write(sa)

PRO: Simple operations
CON: Complexity proportional to the dynamic number of memory references

Both are always applicable, but only LRPD is guaranteed to give an answer for any input!
isIndep = .FALSE. IF (nsymm.EQ.0)
           isIndep = .TRUE. ELSE
           acc_i=.TRUE. PARALLEL DO i=1,10
                       acc_j = .TRUE. DO j=1,10
                       pt = C(j,i).GT.0 acc_j = acc_j .AND. pt
                       ENDDO acc_i = acc_i .AND. acc_j
           ENDDO
           isIndep = acc_i
END
Code Generation: DYFESM / SOLVH_do20

READ *, nsymm
DO step = 1, 1000
  DO i=1, 10
    IF (nsymm.EQ.0) THEN
      WORK(1:10) = ...
    ELSE
      DO j=1,10
        IF (C(j,i).GT.0) THEN
          WORK(j) = ...
        ENDIF
      ENDDO
    ENDIF
  ENDDO
  ... = WORK(1:10)
ENDDO

Hoisting of run-time tests
Dynamic Parallel Coverage

Compile-time
RT: Simple Checks
RT: Sorting
RT: USR Evaluation
RT: LRPD

PERFECT
SPEC2000
Previous SPEC
Performance Results: Speedup

![Graph showing performance speedup for different applications and processor configurations. The x-axis represents different applications, and the y-axis represents speedup. The graph compares 1 Processor, 2 Processors, 4 Processors, and 4 Processors (CT only). The applications are categorized into PERFECT, SPEC2000, and Previous SPEC.]
Hybrid Analysis Applications

- Array Data Flow Analysis
  - $Use = USR_1$
  - $Def = USR_2$
  - $UseDef$ edge weight $= USR_1 \cap USR_2$
  - No flow $\iff USR_1 \cap USR_2 = \emptyset$

- Locality Enhancement
  - $\bigcup_{i=1, \text{tile\_size}} (USR_i) \subseteq \text{Level 2 Cache}$

- Checkpointing
  - Exclusion of dead or read-only memory

Parasol
Conclusions

- Hybrid memory reference and dependence analysis
  - USR
    - Closed-form representation that tolerates analysis failure
  - PDAG:
    - Input sensitivity of optimization decisions
    - Continuum of compile-time to run-time solutions

- Efficient automatic parallelization
  - Speedups on Fortran 77 benchmark applications