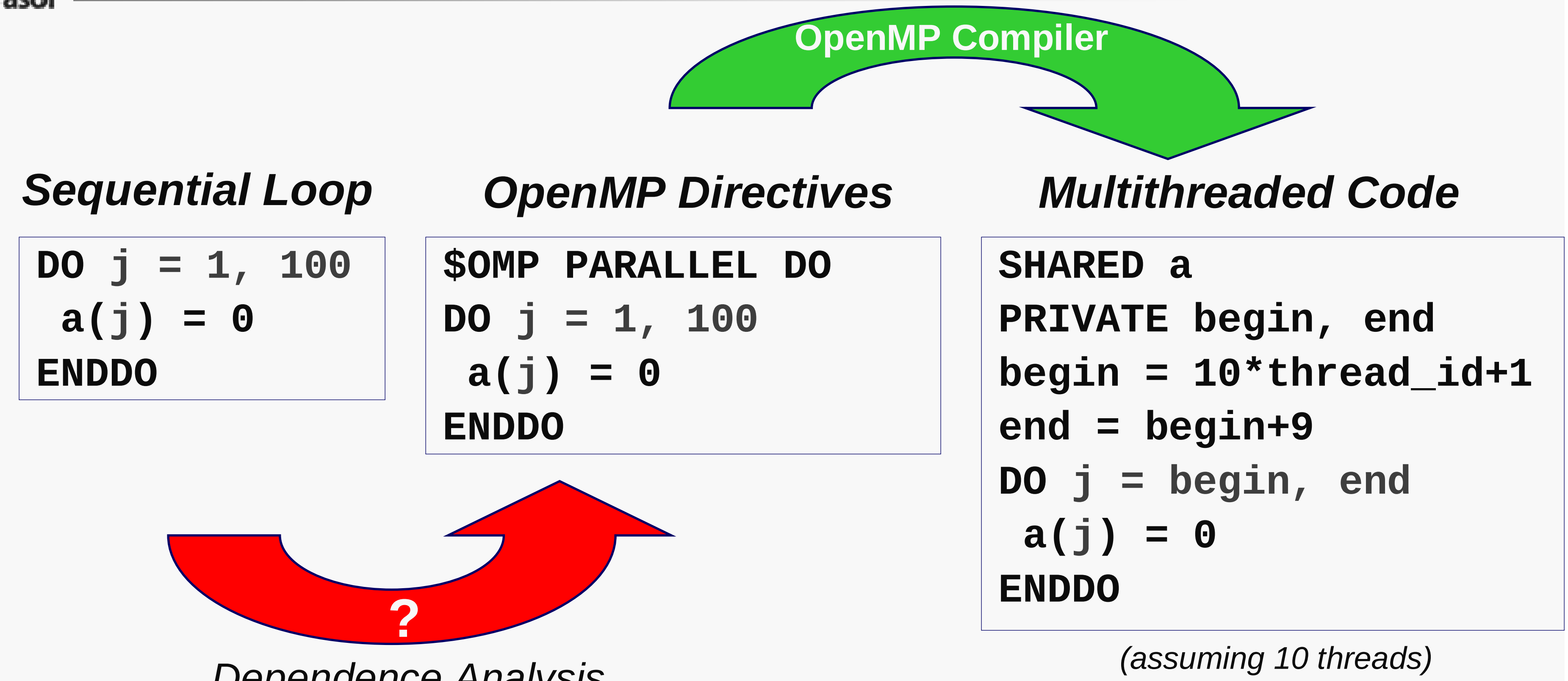
Hybrid Analysis
and its Application to Automatic Parallelization

Silvius Rus

Lawrence Rauchwerger

Parasol
Smarter computing.
Texas A&M University

Thread Level Loop Parallelization



Sequential Loop

```
DO j = 1, 100
 a(j) = 0
ENDDO
```

OpenMP Directives

```
$OMP PARALLEL DO
DO j = 1, 100
 a(j) = 0
ENDDO
```

Multithreaded Code

```
SHARED a
PRIVATE begin, end
begin = 10*thread_id+1
end = begin+9
DO j = begin, end
 a(j) = 0
ENDDO
```

(assuming 10 threads)

Dependence Analysis

Privatization (renaming)

Reduction parallelization

…

Data Dependence

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

X = …	… = X	X = …
… = X	X = …	X = …
flow	**anti**	**output**

Loop Parallelization

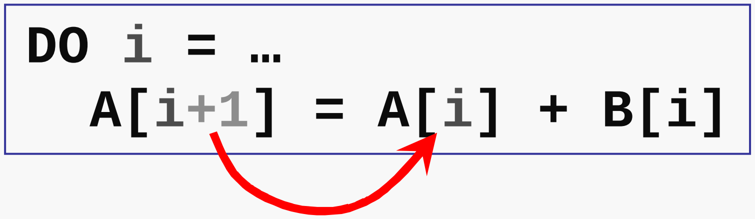
Can a loop be executed in parallel ?

- 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

```
DO i = …
  A[i] = A[i] + B[i]
```

A sequential loop



```
DO i = …
  A[i+1] = A[i] + B[i]
```

Static Data Dependence Analysis

Dependence set = solutions to a system of linear inequations

Independence ⟺ no integer solutions

GCD, Banerjee, Range Test, Omega

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

No cross iteration dependences ⟺ ***No integer solutions:***

$$\begin{cases} 1 \le j_1 \le 10 \\ 1 \le j_2 \le 10 \\ j_1 \ne j_2 \\ j_1 = j_2 + 40 \end{cases}$$

Input-Sensitive Decisions

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

WRITE ***READ***

`N =  5: [1: 5] ∩ [41:45] =    Ø` **Independent**

`N = 45: [1:45] ∩ [41:85] = [41:45]` **Dependent**

Different outcomes depending on the value of **`N`**.

Compile-time analysis fails!

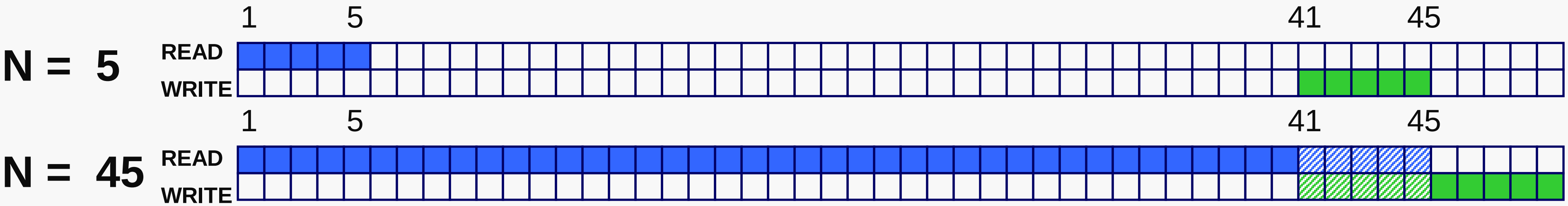
```
READ *, N
DO j=1,N
 a(j)=a(j+40)
ENDDO
```

Run-time Analysis

LRPD Test

1. Instrumentation of all relevant memory references

2. Run-time analysis of the resulting trace



Predicate Extraction

Extract conditions under which there are no dependencies

```
READ *, N
IF (N.LE.40) THEN
   Parallel Loop
ELSE
   Sequential Loop
ENDIF
```

Only simple cases!

A Motivating Example

```
READ *, N, offsets(1:N)
DO j = 1, N
 ind = offsets(j)
 DO i = 1, N
  DO k = 1, N
   ind = ind+1
   a(ind) = …
  ENDDO
 ENDDO
ENDDO
```

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

Compile Time

- PROs
  - No run-time overhead
- CONs: too conservative
  - Dependence on input values
  - Weak symbolic analysis
    - Subscripted subscripts
    - Complex recurrences
    - Address-data computation cycles
  - Impractical symbolic analysis
    - Combinatorial explosion

Run-time, reference by reference

- PROs
  - Always finds answers
- CONs
  - Run-time overhead proportional to dynamic memory reference count
  - Ignores partial compile-time analysis results

Hybrid Analysis of Memory Reference Patterns

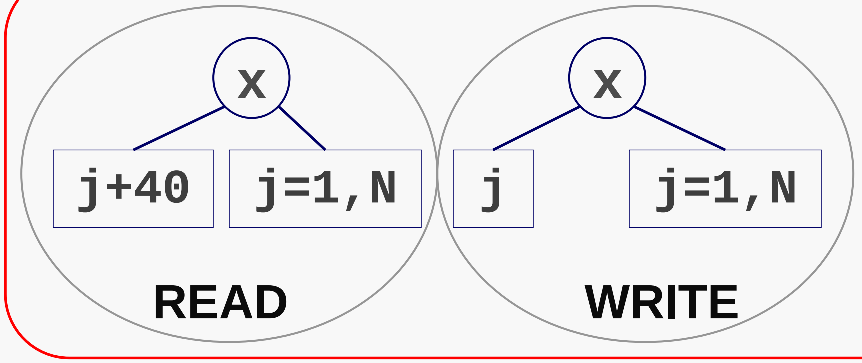
	Compile-time Analysis	Hybrid Analysis	Run-time Analysis
STATIC	Symbolic analysis	Symbolic analysis	
DYNAMIC		Continue analysis with actual values	Full reference-by-reference analysis

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?

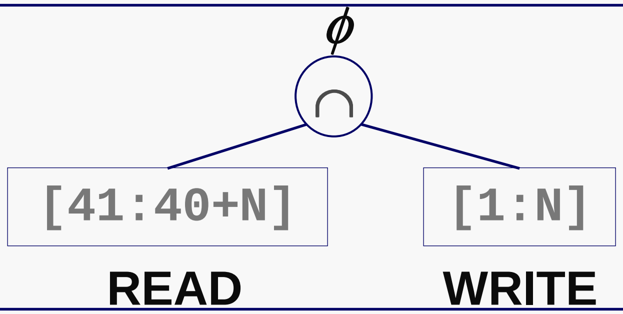


1. Collect references.

`[41:40+N]` `[1:N]`

READ **WRITE**

2. Aggregate them symbolically.



3. Formulate independence test.

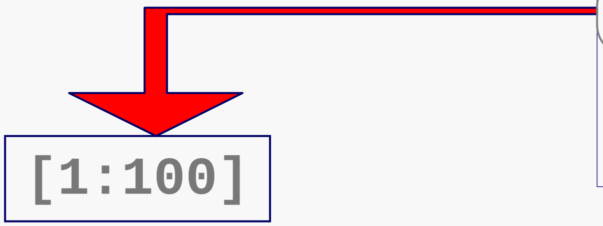
`N.LE.40`

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**

```
[1:100]
```

```
SUBROUTINE Rfft(cc, ch)
INTEGER na, cc(*), ch(*)
ch(1:100)=d(1:100)
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
```

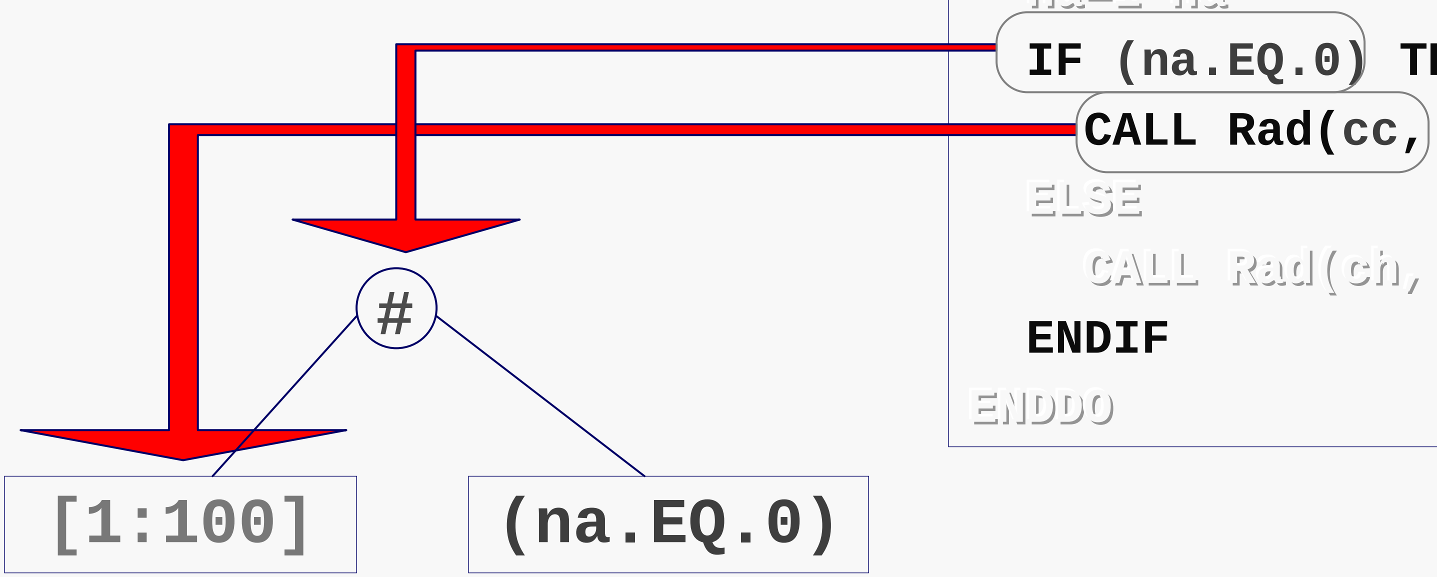
- This case also solved at compile-time

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**

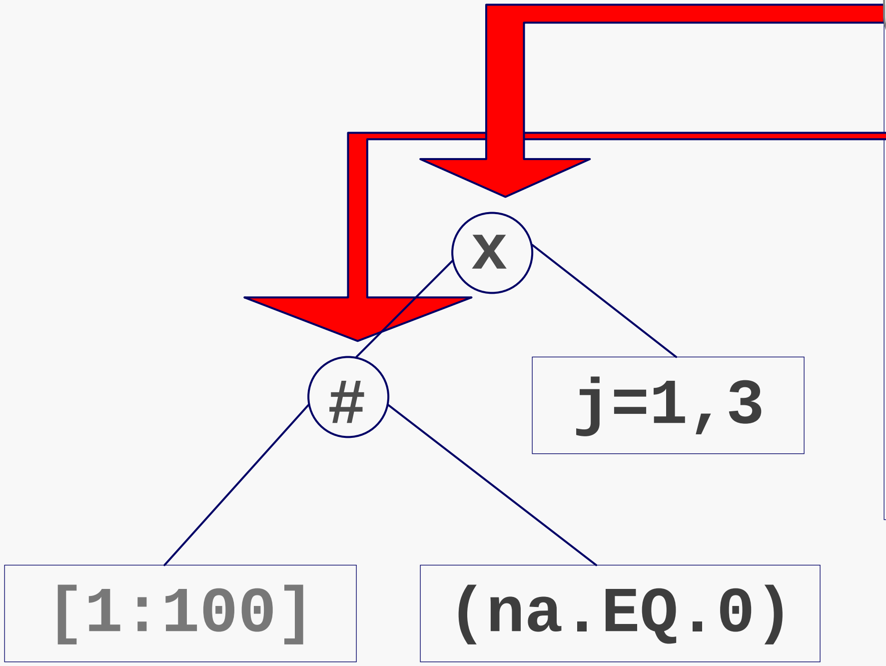
```
SUBROUTINE Rfft(cc, ch)
INTEGER na, cc(*), ch(*)
na=1
DO j=1,3
 na=1-na
 IF (na.EQ.0) THEN
  CALL Rad(cc, ch)
 ELSE
  CALL Rad(ch, cc)
 ENDIF
ENDDO
```



Recurrence Operator

Postpone Analysis Failure due to a Recurrence with no Closed Form

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

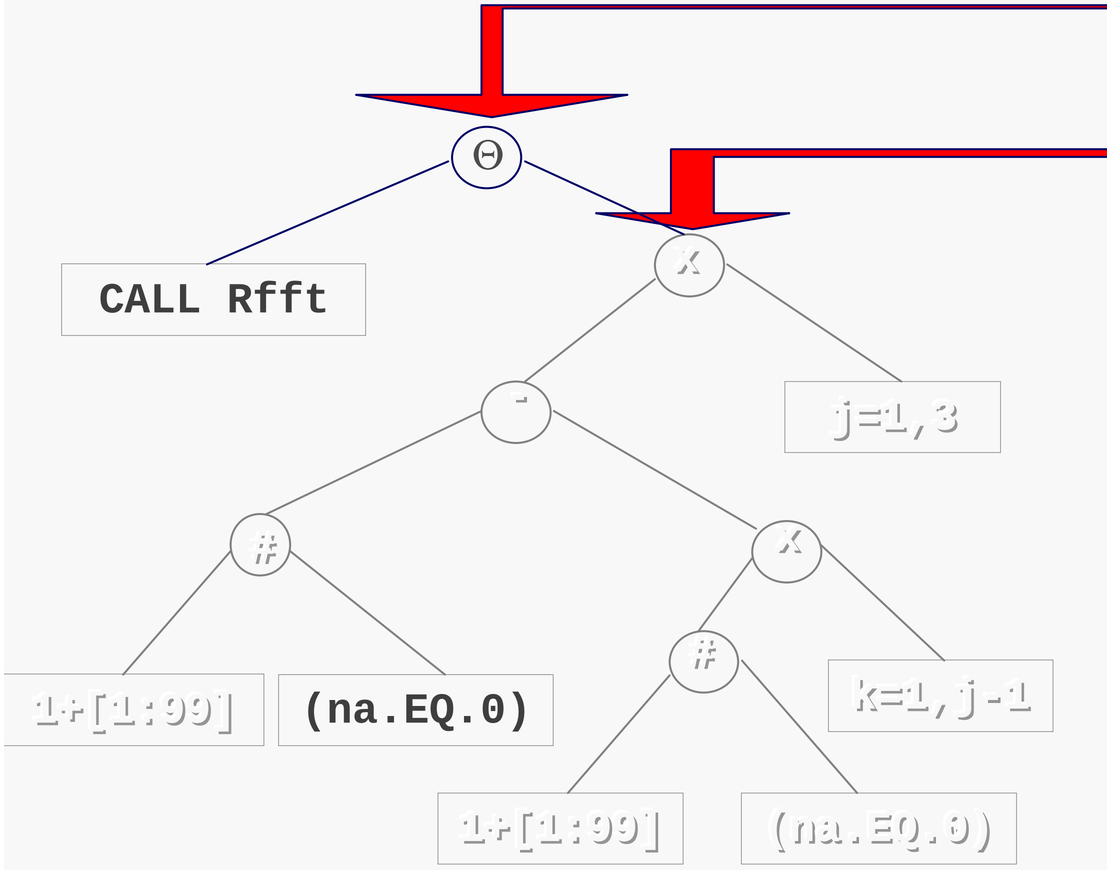


```
SUBROUTINE Rfft(cc, ch)
INTEGER na, cc(*), ch(*)
na=1
DO j=1,3
  na=1-na
  IF (na.EQ.0) THEN
    CALL Rad(cc, ch)
  ELSE
    CALL Rad(ch, cc)
  ENDIF
ENDDO
```

Translation Operator

Postpone Analysis Failure due to Translation Issues (reshaping etc)

```
SUBROUTINE Run(w, d)
INTEGER w(*), d(100,*)
DO j=1,1000
  CALL Rfft(w, d(1,j))
ENDDO
```



```
SUBROUTINE Rfft(cc, ch)
INTEGER na, cc(*), ch(*)
ch(1:100)=d(1:100)
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
```

WF pattern on array `w`

`na` - local to `Rfft`

Uniform Set of References (USR)

T = { **LMAD**, $\cap$, $\cup$, $-$, **(**, **)**, **#**, **x**, Θ, **Gate**, **Recurrence**, **Call Site**}

N = { USR }

S = USR

P = { USR $\rightarrow$ **LMAD** | **(**USR **)**
USR $\rightarrow$ USR $\cap$ USR
USR $\rightarrow$ USR $\cup$ USR
USR $\rightarrow$ USR $-$ USR
USR $\rightarrow$ USR # **Gate**
USR $\rightarrow$ USR **x Recurrence**
USR $\rightarrow$ USR Θ **Call Site** }

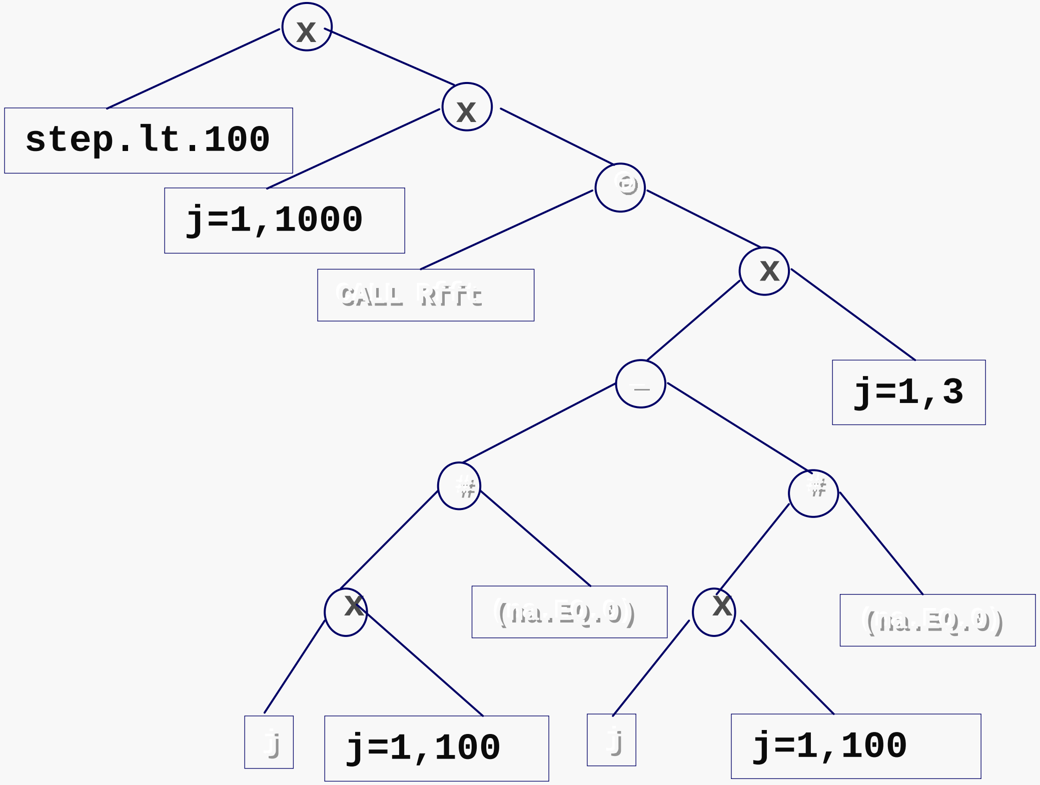
LMAD = Start + [Stride$_1$:Span$_1$, Stride$_2$:Span$_2$, ...]

Reducing Complexity

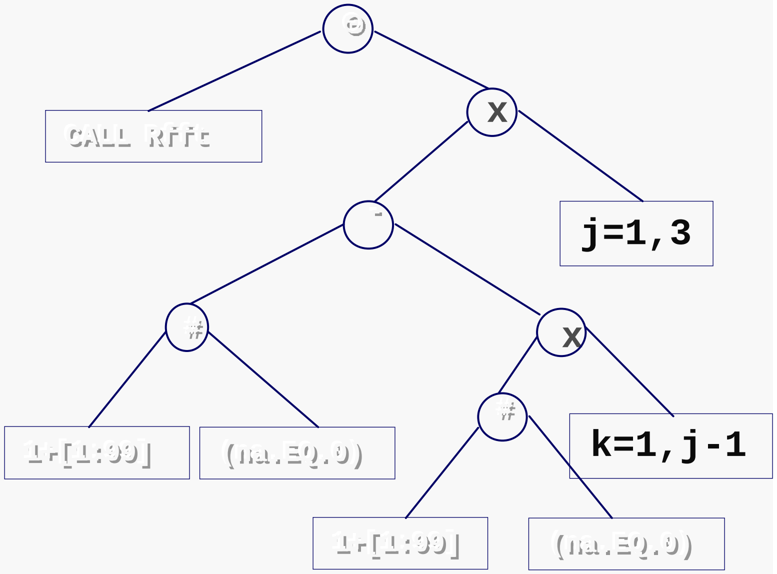
- At compile-time
  - Contiguous aggregation, interleaving of LMADs
  - Loop invariant USR hoisting
  - Logic inference, e.g. $G\#D_1 \cap \overline{G}\#D_2 = \varnothing$
  - Set identities, e.g. $(A - B) - A = \varnothing$
  - Lattice properties, e.g. $A - \mathrm{T} = \varnothing$

- At run-time
  - Contiguous aggregation, interleaving

Reference-by-reference pure RT test



Hybrid Analysis



Number of iterations necessary for computing access pattern

100 * 1000 * 3 * 100 = 30,000,000

3

The USR's Contribution:
Static Analysis Failure Tolerance

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

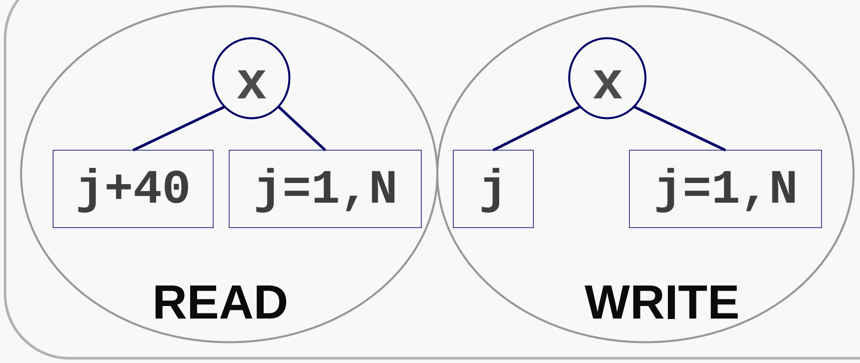
- Previous representations
  - Are limited to
    - Linear subscripts, control predicates, loop bounds
    - Direct indexing only (no subscripted subscripts)
  - At points of failure
    - Stop analysis or
    - Approximate (often overly) conservatively

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

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

Are there any cross-iteration dependences?

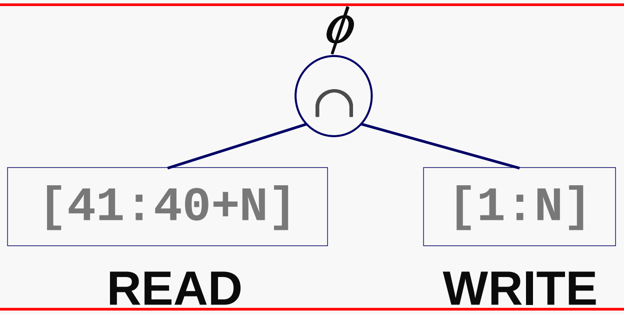


1. Collect references.

`[41:40+N]` **READ**

`[1:N]` **WRITE**

2. Aggregate them symbolically.



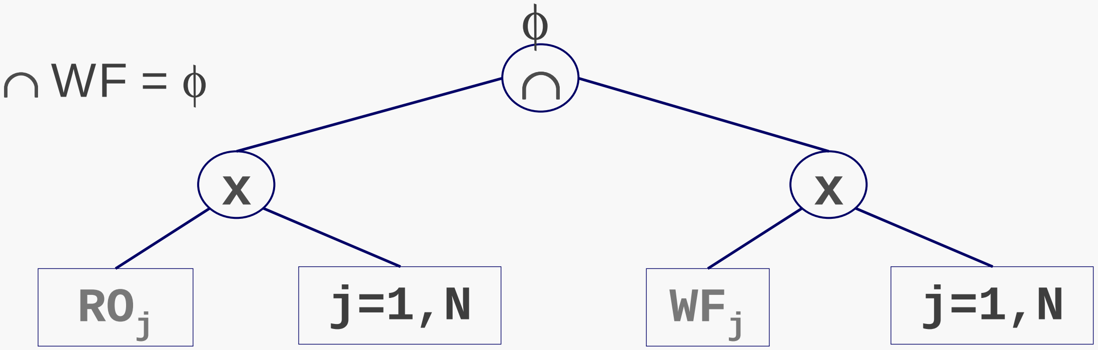
3. Formulate independence test.

`N.LE.40`

4. Extract lowest-cost runtime test.

Data Dependences

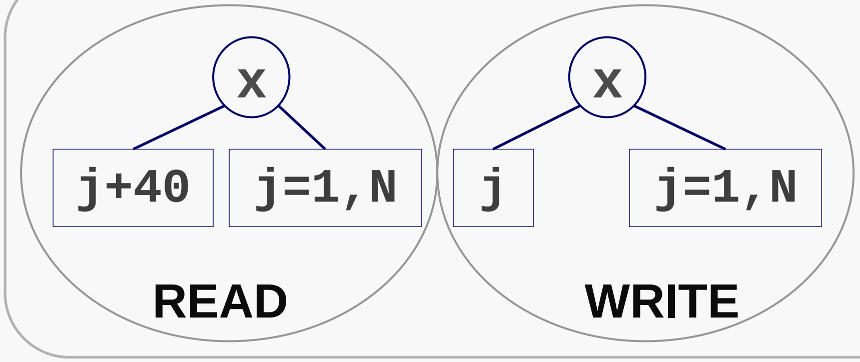
- Given:
  - Loop expression: **`j = 1,N`**
  - Per-iteration aggregated descriptors $\mathtt{RO}_j$, $\mathtt{WF}_j$, $\mathtt{RW}_j$
- Solve equation RO $\cap$ WF = ϕ



- 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

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

Are there any cross-iteration dependences?

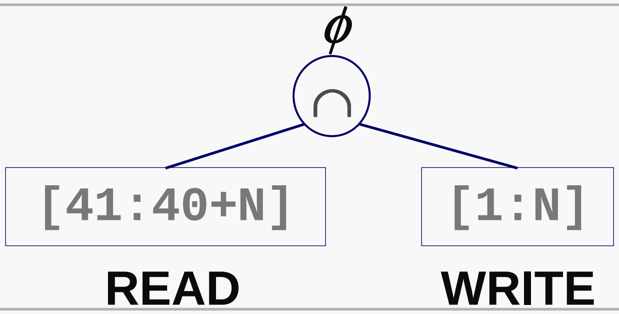


1. Collect references.

`[41:40+N]` `[1:N]`

READ **WRITE**

2. Aggregate them symbolically.



3. Formulate independence test.

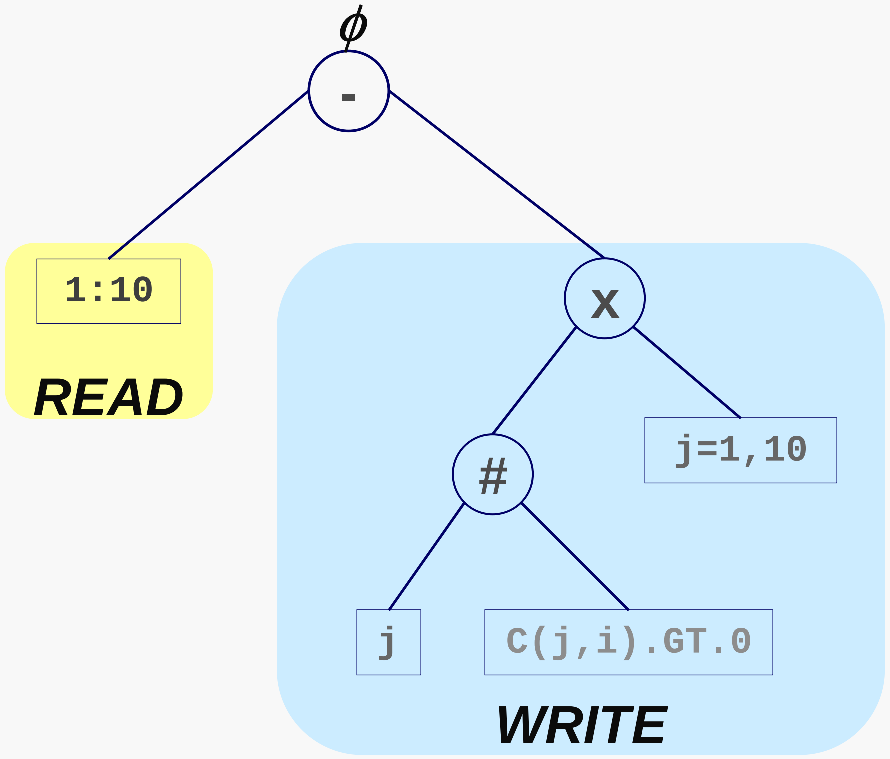
`N.LE.40`

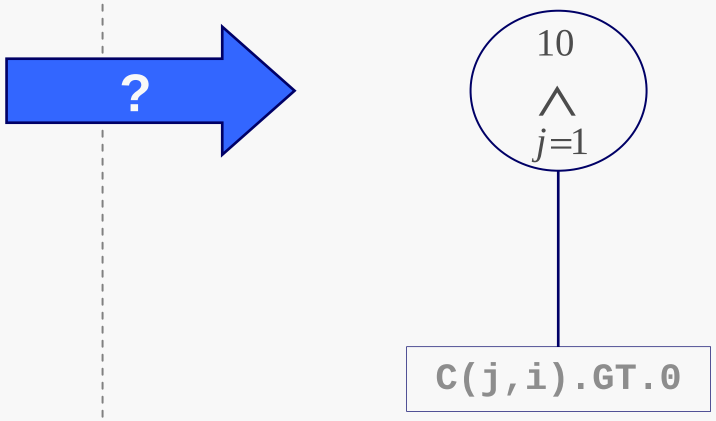
4. Extract lowest-cost runtime test.

Example:
Exposed Read

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

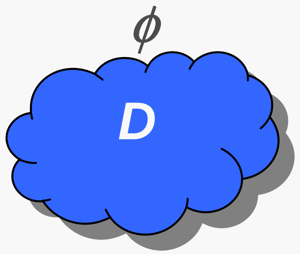
USR Equation





Predicate Tree

Proof System



Input: ***USR equation*** $D = \phi$

Output: ***Predicate*** P

Such that:

$$P \Longleftrightarrow D = \phi$$

OPTIMISTIC: Sufficient predicate: $P \Rightarrow D = \phi$

PESSIMISTIC: Necessary predicate: $D = \phi \Rightarrow P$, ***or*** $\bar{P} \Rightarrow D \neq \phi$

How Hard Is This Problem?

```
A(1) = …
j = f(x)
A(j) = …
```

$$j \overset{?}{=} 1$$

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

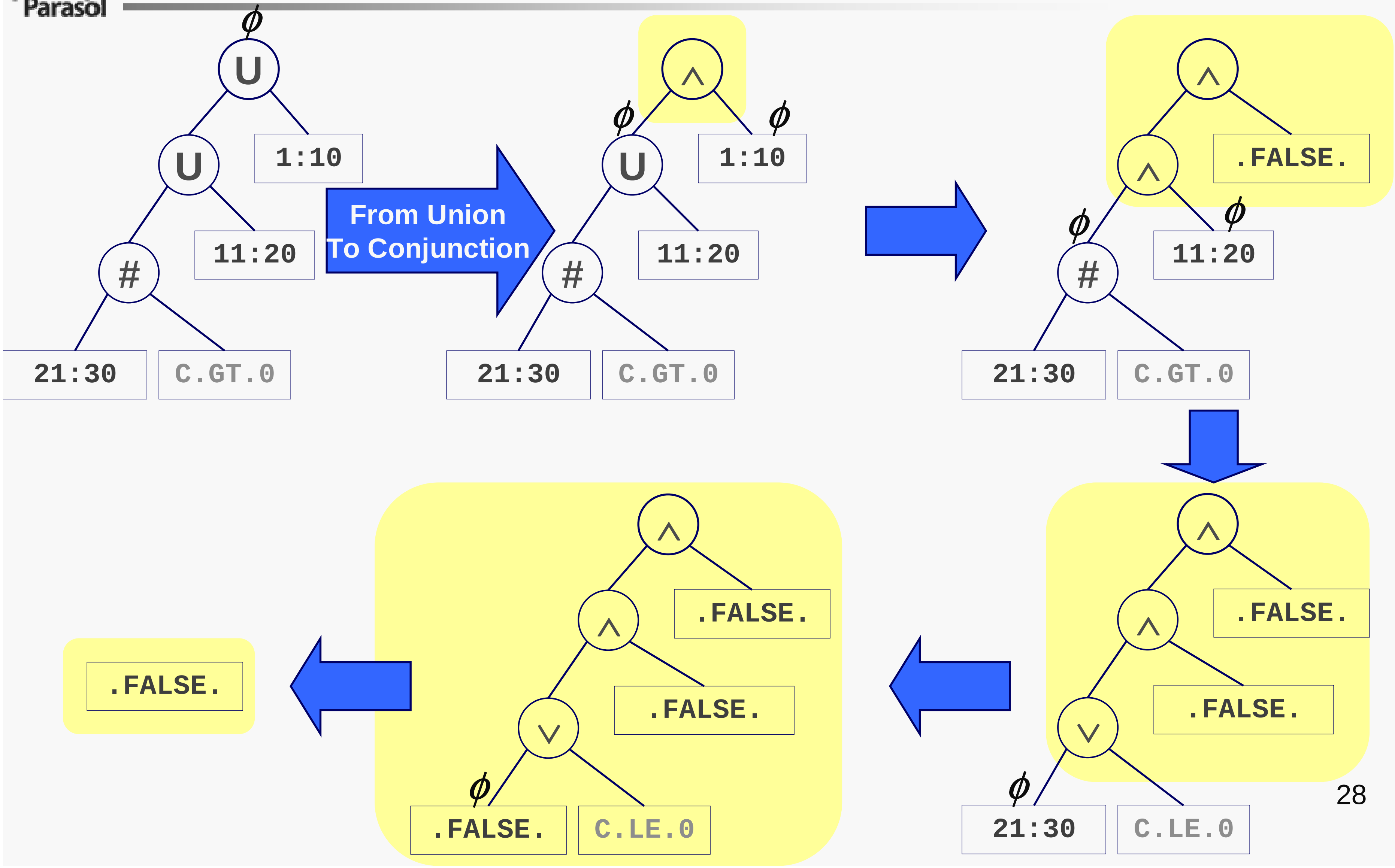
$$\mathtt{P(f(x)) = true} \Leftrightarrow \mathtt{x} \in \mathtt{f}^{-1}(\mathtt{P}^{-1}(\mathtt{true}))$$

Worst case: $O(2^{\mathrm{size}(x)} * \mathrm{complexity}(P \bullet f))$

But most real cases are tractable!

Our Solution:

Recursive Descent on the USR Tree



Predicate Formal Specification

T = { Logical Expression, $\wedge$, $\vee$, $\otimes_\wedge$, $\otimes_\vee$, Θ, Recurrence, Call Site, Library Routine}

N = { PDAG }

S = PDAG

P = { PDAG $\rightarrow$ Logical Expression
PDAG $\rightarrow$ PDAG $\wedge$ PDAG
PDAG $\rightarrow$ PDAG $\vee$ PDAG
PDAG $\rightarrow$ PDAG $\otimes_\wedge$ Recurrence
PDAG $\rightarrow$ PDAG $\otimes_\vee$ Recurrence
PDAG $\rightarrow$ PDAG Θ Call Site
PDAG $\rightarrow$ Library Routine }

Grammar-directed Translation: Algorithm *Solve*

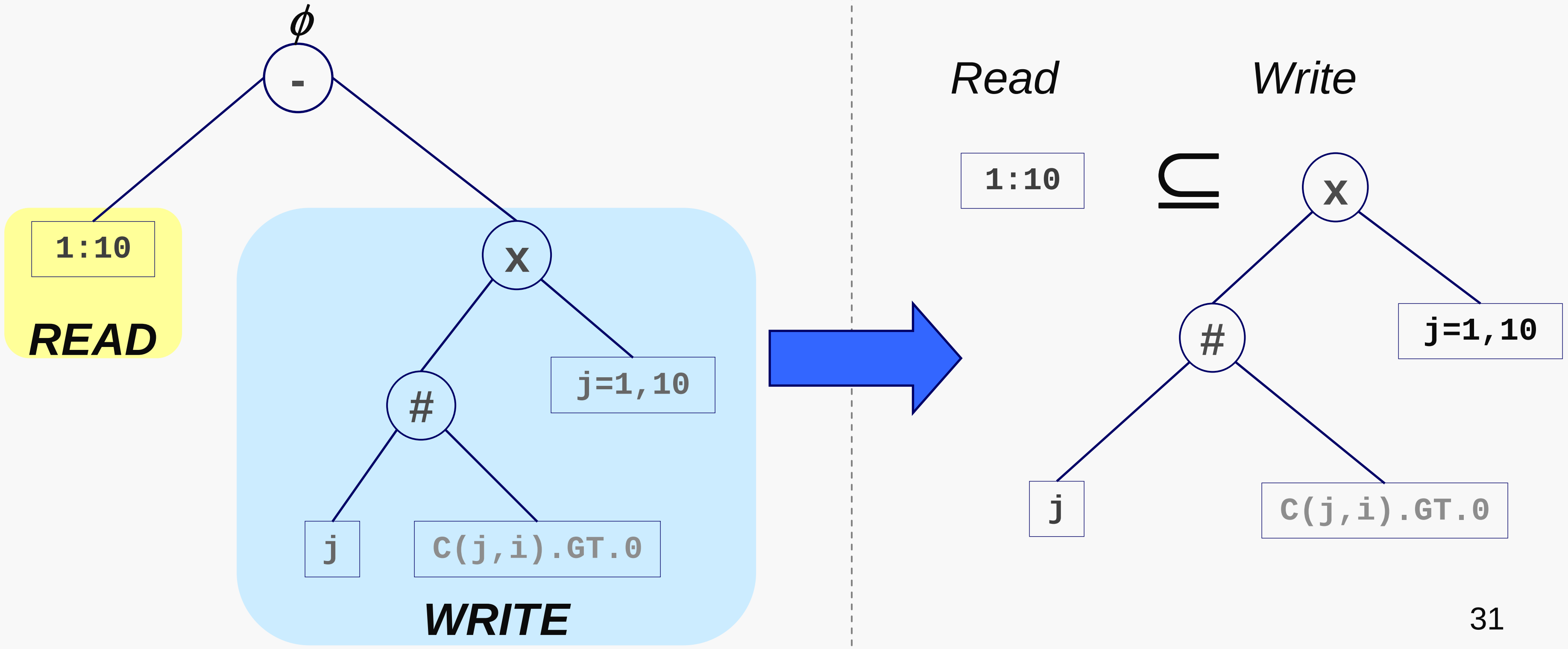
- Input: D=∅ (AST on USR grammar)
- Output: P (AST on PDAG grammar)

```
CASE root(D) OF
  Leaf:                          $P = false$
  Union(A,B):                    $P = Solve(A=\phi) \wedge Solve(B=\phi)$
  Intersection(A,B):             $P = Solve(A=\phi) \vee Solve(B=\phi) \vee Solve\ Disjoint(A, B)$
  Difference(A,B):               $P = Solve(A=\phi) \vee Solve\ Inclusion(A, B)$
  Predicate(p,A):                $P = \overline{p} \vee Solve(A=\phi)$
  Loop Expansion(A_i, i=1,N):    $P = \prod_{i=1,N} Solve(A_i)$
  Translation(CallSite, A):      $P = Translate(CallSite, Solve(A=\phi))$
IF (P not equivalent to (D=∅)) Then $P = P \vee$ Reference Based Test(D=∅)
```

Inclusion Example:
Exposed Read

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

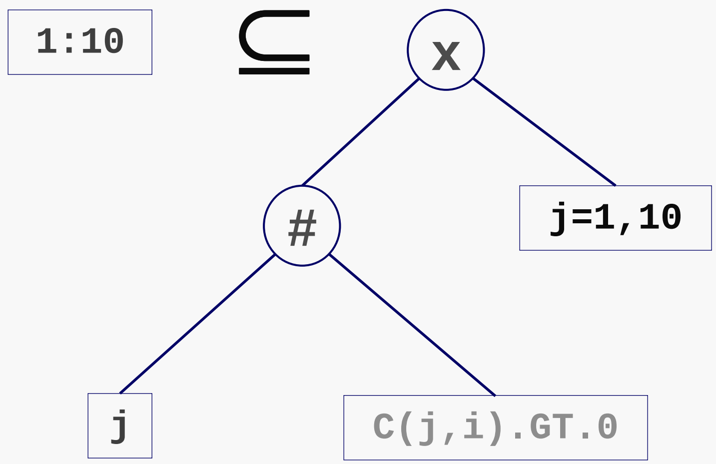
USR Equation



Inclusion Example

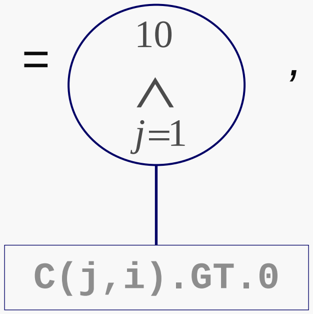
```
DO i=1, 10
  DO j=1,10
    IF (C(j,i).GT.0) THEN
      WORK(j) = …
    ENDIF
  ENDDO
  … = WORK(1:10)
ENDDO
```

Read *Write*



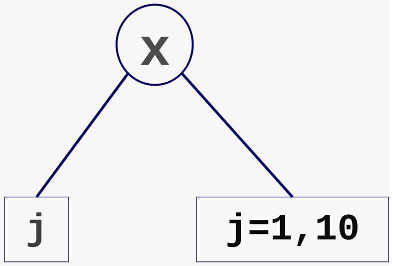
Extract predicate p_{WRITE} under which *WRITE* is maximal.

p_{WRITE} =

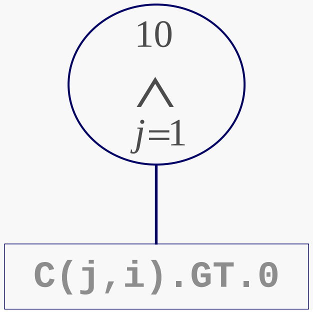


,

WRITE =



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



Algorithm *Solve Inclusion*

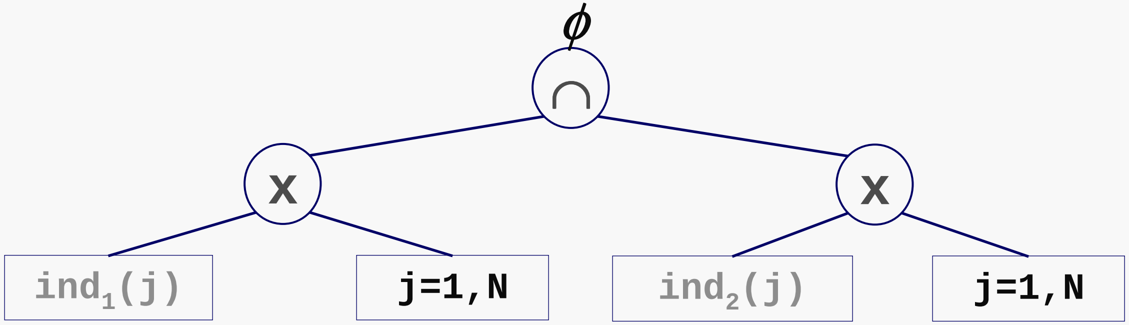
- Input: A,D (AST on USR grammar)
- Output: P (AST on PDAG grammar)

```
CASE root(D) OF
  Union(B,C):          $P = \text{Solve}(A-B=\phi) \vee \text{Solve}(A-C=\phi) \vee$ ? (see ✡)
  Intersection(B,C):   $P = \text{Solve}(A-B=\phi) \wedge \text{Solve}(A-C=\phi)$
  Difference(B,C):     $P = \text{Solve}(A-B=\phi) \vee \text{Solve}\ (A \cap C)$
  Predicate(q,B):      $P = q \wedge \text{Solve}(A-B=\phi)$
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, $<q_A, \lceil A \rceil>$
  Extract maximal (closest) predicated underestimate of D, $<q_D, \lfloor D \rfloor>$
  $P = q_A \wedge q_D \wedge$ Solve Inclusion LMADs$(\lceil A \rceil, \lfloor D \rfloor)$
```

Similar for Solve Disjoint

Fallback

Not all equations can be reversed



ϕ

$\cap$

x

x

$\mathtt{ind_1(j)}$

`j=1,N`

$\mathtt{ind_2(j)}$

`j=1,N`

Solutions

Pattern library

Monotonic ***Injective***

…

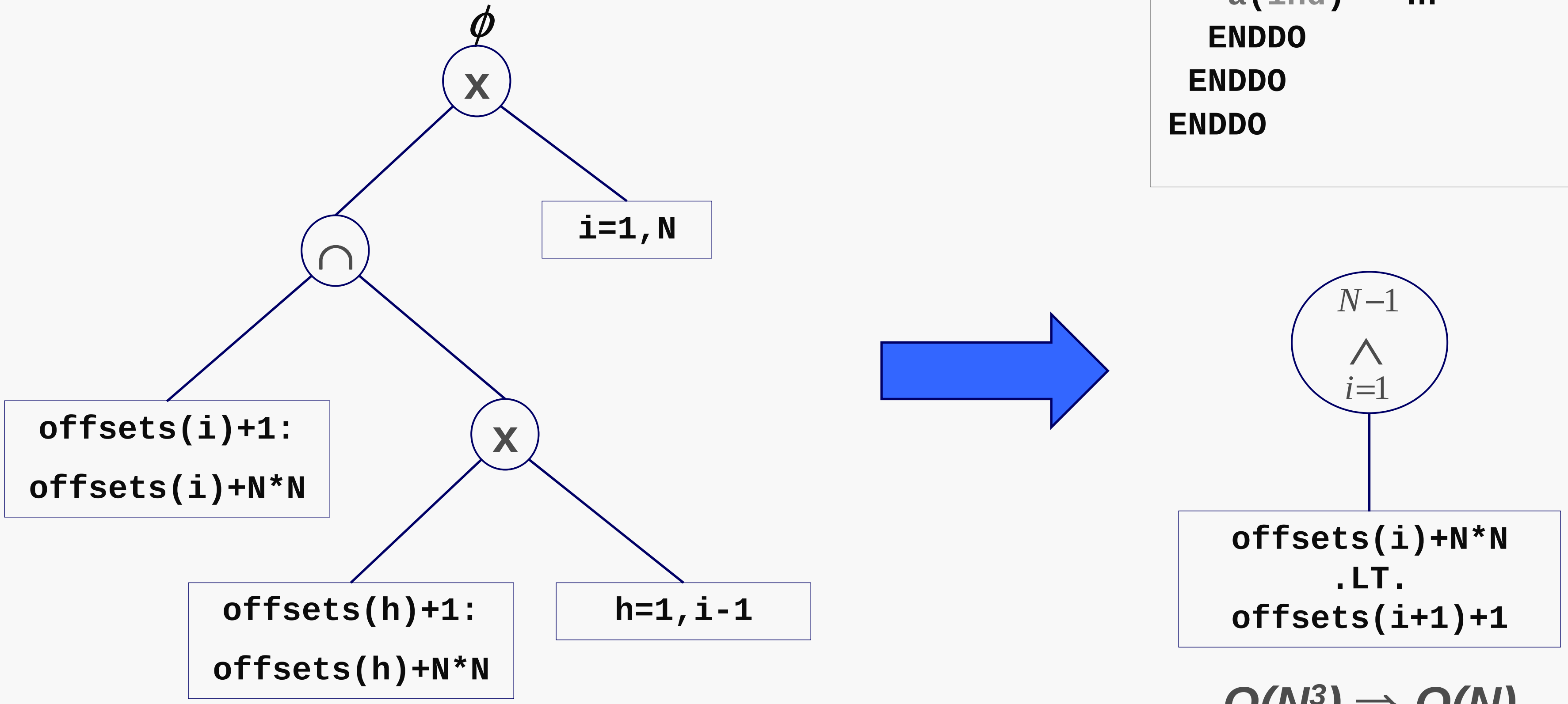
Extensible Compiler!

Reference-based test

LRPD ***USR***

Monotonicity

```
READ *, N, offsets(1:N)
DO i = 1, N
 ind = offsets(i)
 DO j = 1, N
  DO k = 1, N
   ind = ind+1
   a(ind) = …
  ENDDO
 ENDDO
ENDDO
```



ϕ

x

∩

`i=1,N`

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

x

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

`h=1,i-1`

$\bigwedge_{i=1}^{N-1}$

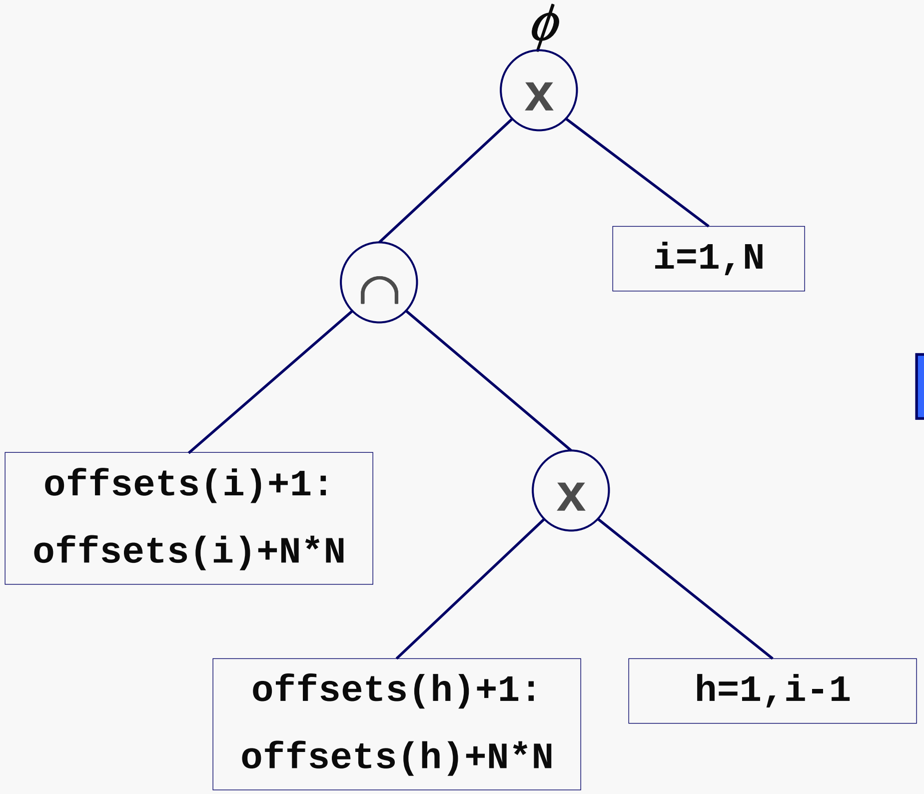
`offsets(i)+N*N`
`.LT.`
`offsets(i+1)+1`

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

Sufficient, but
not necessary

Injectivity

```
READ *, N, offsets(1:N)
DO i = 1, N
 ind = offsets(i)
 DO j = 1, N
  DO k = 1, N
   ind = ind+1
   a(ind) = …
  ENDDO
 ENDDO
ENDDO
```



ϕ

x

∩

`i=1,N`

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

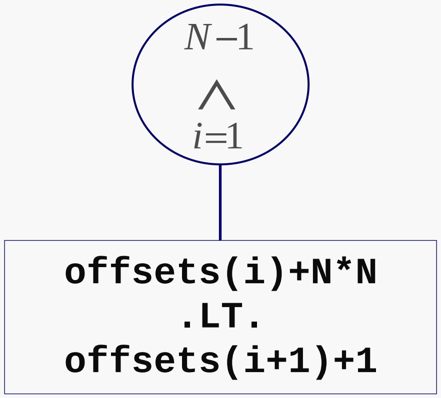
x

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

`h=1,i-1`

1. Sort(`offsets(1:N)`)

2.



$\bigwedge_{i=1}^{N-1}$

`offsets(i)+N*N`
`.LT.`
`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

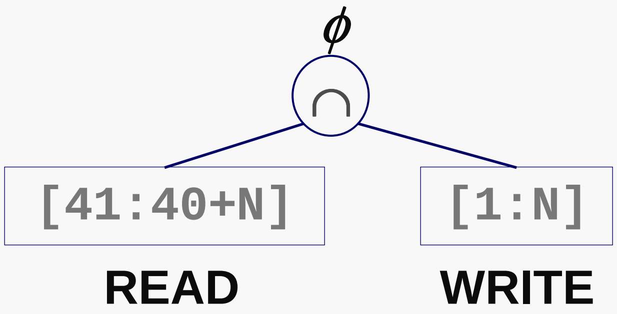
- Pattern recognition
  - USR equivalence rules

- Canned solutions
  - Data dependence: monotonicity, sorting
  - Other uses: specific patterns

Reference-based Runtime Tests

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

Aggregated USR Evaluation


ϕ

$\cap$

[41:40+N]

[1:N]

READ

WRITE

```
CALL build_USR(41, 40+N, D0)
CALL build_USR(1, N, D1)
CALL intersect(D0, D1, D2)
isIndependent = check_empty(D2)
```

USR → FORTRAN: attribute grammar

PRO: May reduce asymptotic complexity

CON: Uses expensive operations

Reference by reference LRPD

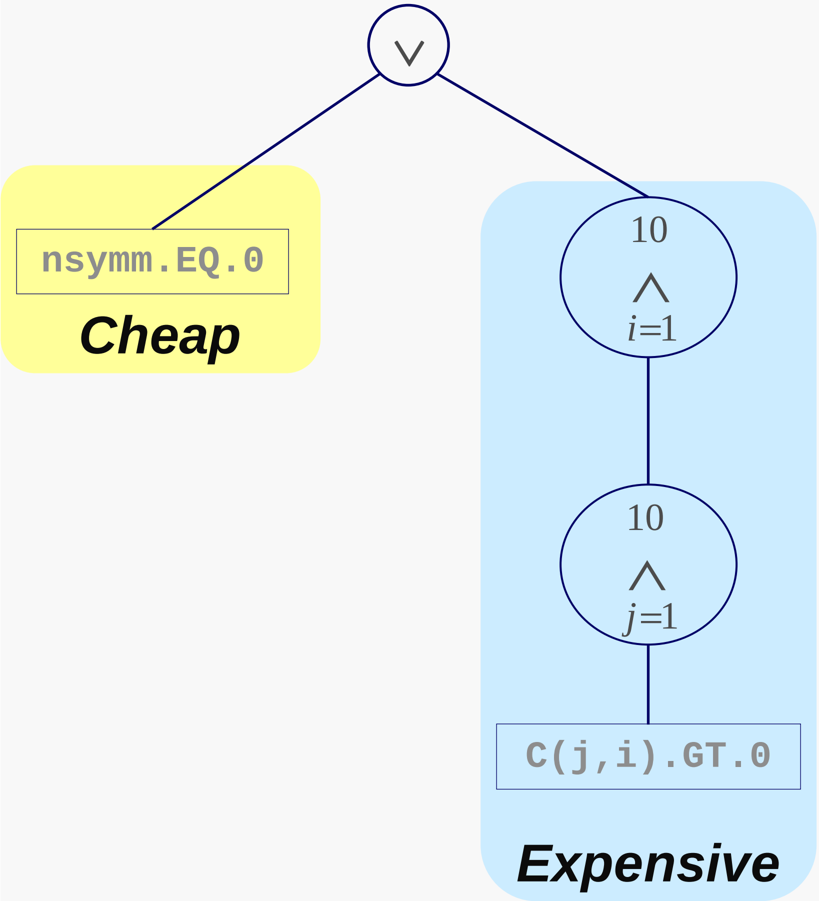
```
DIMENSION a(100)
DIMENSION sa(100)
…
DO j=1,N
  CALL mark_write(sa, j)
  CALL mark_read(sa, j+40)
ENDDO
isIndependent =
 is_disjoint_read_write(sa)
```

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!

Code Generation: DYFESM / SOLVH_do20

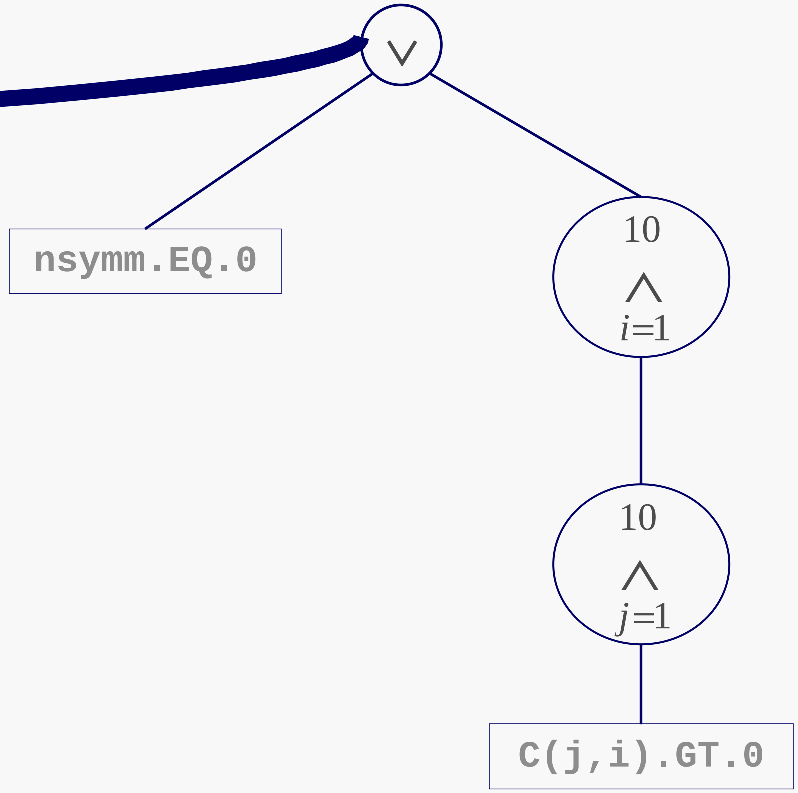


```
isIndep = .FALSE.
IF (nsymm.EQ.0)
 isIndep = .TRUE.
ELSE
 acc_i=.TRUE.
 PARALLEL DO i=1,10
  acc_j = .TRUE.
  DO j=1,10
   pt = C(j,i).GT.0
   acc_j = acc_j .AND. pt
  ENDDO
  acc_i = acc_i .AND. acc_j
 ENDDO
 isIndep = acc_i
END
```

Cascade of tests in increasing complexity order

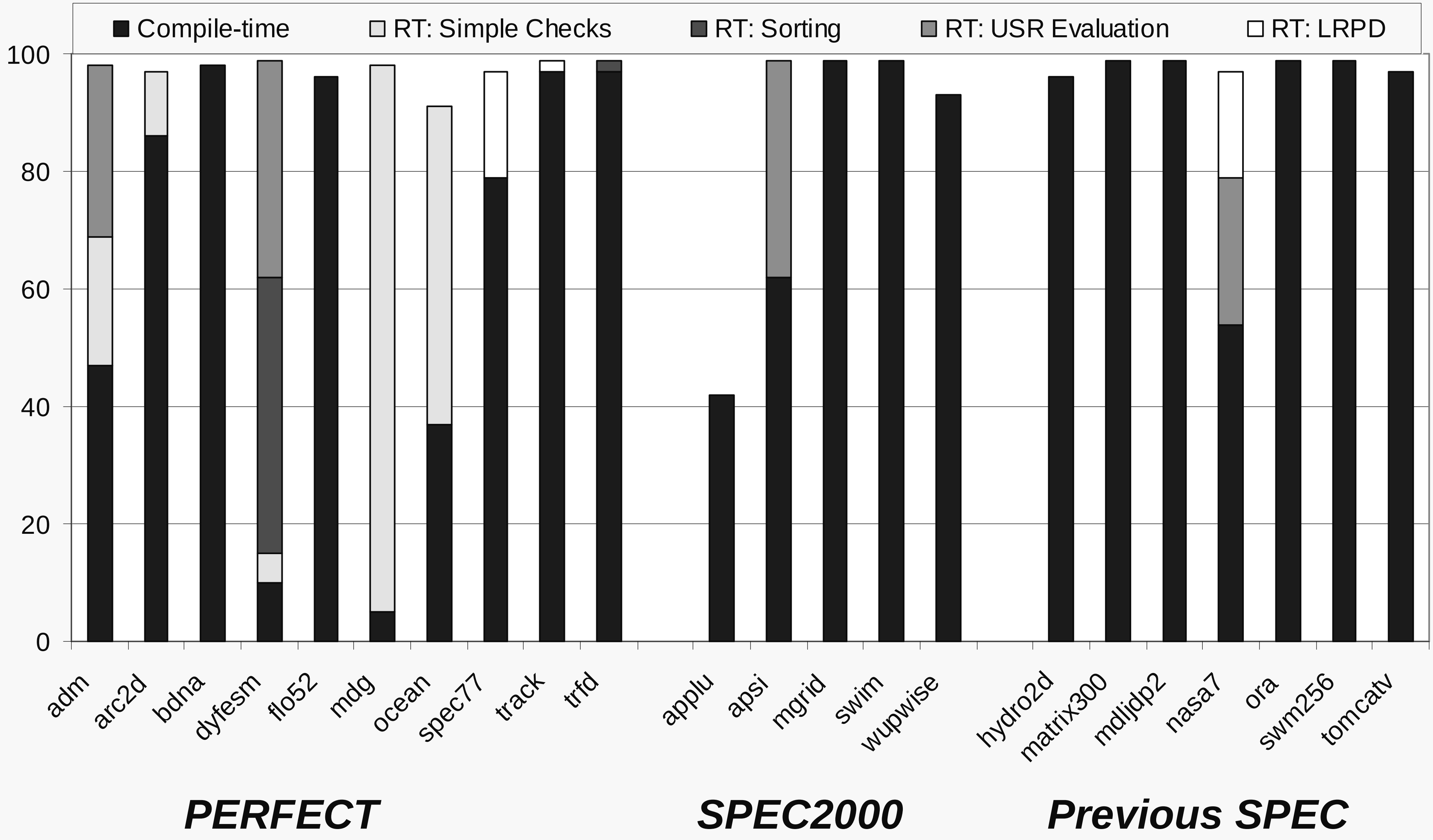
Code Generation: DYFESM / SOLVH_do20

```
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
    … = WORK(1:10)
  ENDDO

ENDDO
```

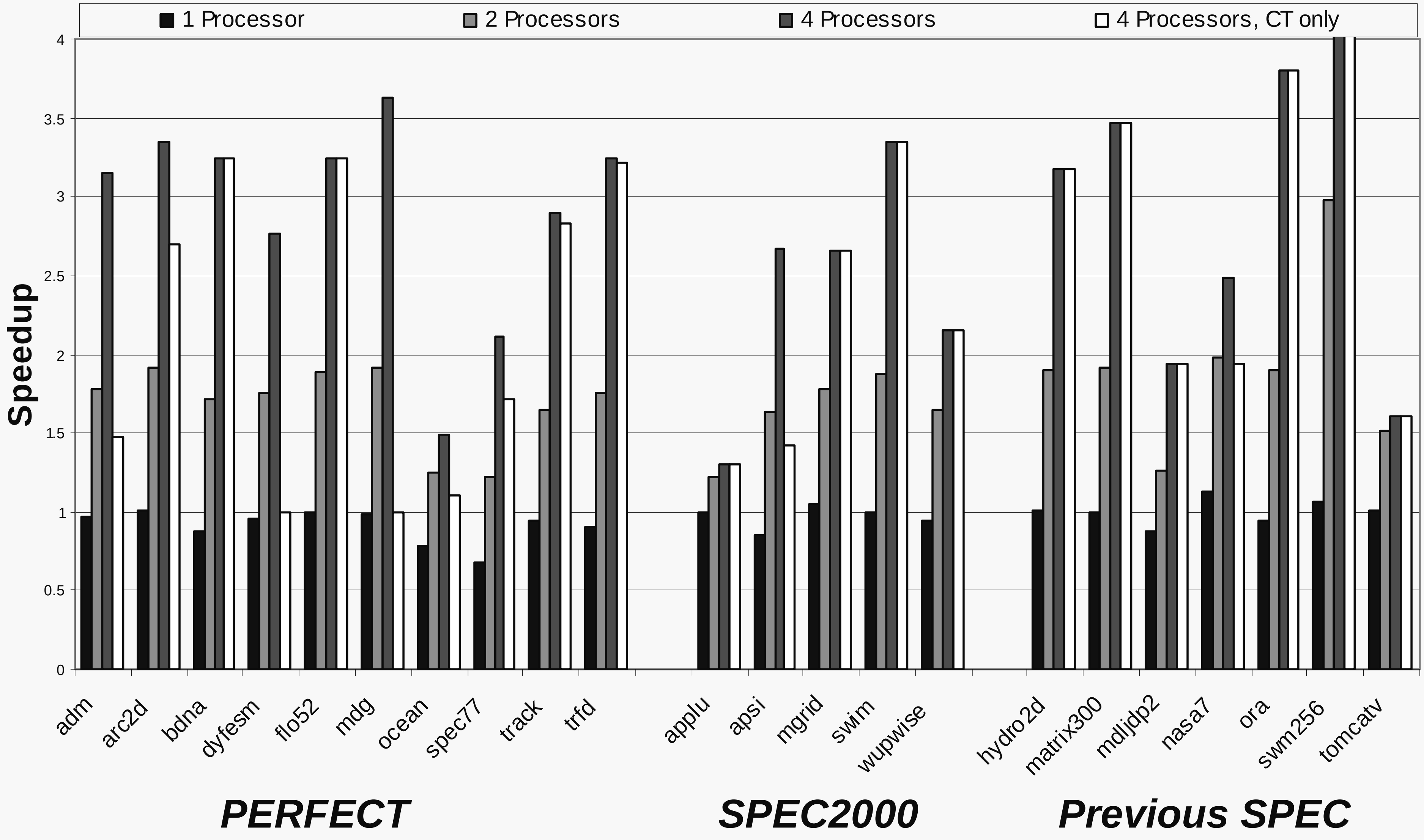


Hoisting of run-time tests

Dynamic Parallel Coverage



Performance Results: Speedup



Hybrid Analysis Applications

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

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

- Checkpointing
    - Exclusion of dead or read-only memory

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