Hybrid Analysis of Memory Reference Patterns

Silvius Rus
Lawrence Rauchwerger
Jay Hoeflinger

Texas A&M University
Texas A&M University
Intel - KAI Software Lab
Hybrid Analysis

- **Static analysis:**
  - No overhead
  - Inexact

- **Run-time analysis:**
  - Exact
  - Expensive

*Hybrid* analysis: exact and inexpensive
Find intersection of **READ** and **WRITE** memory access patterns (find dependencies)

Memory Reference Set as a tree

Memory reference expressions and program constructs as **leaves**

Operators as internal **nodes**

Different outcomes depending on \( N \):

- \( N = 5 \): \([41:45] \cap [1:5] = \emptyset\) **PARALLEL**
- \( N = 45 \): \([41:85] \cap [1:45] = [41:45]\) **SEQUENTIAL**
Compile-time Analysis

**Strategy:**
- Evaluate intersection symbolically
- Reduce tree to an aggregated descriptor

Conservative decision: loop is sequential

N - compile-time unknown

```
READ *, N
DO j=1,N
   a(j) = a(j+40)
ENDDO
```
Compile-time Analysis

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

```fortran
READ *, N
DO j=1,N
  a(j)=a(j+40)
  A(j(ind(j))) = ...
  A(=jA) = ...
  A(j+1) = ...
```
Run-time Analysis

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

**Strategy:**
- Evaluate intersection entirely at run-time
- Reduce the tree to a node at run-time

**O(N) run-time overhead**
Run-time Analysis

PROs
Always finds complete answers

CONs
Run-time overhead proportional to dynamic memory reference count

Partially redundant with compile-time analysis
– Cannot use partial results from static analysis
Hybrid Analysis

Strategy:
- Evaluate intersection symbolically
- Prune tree producing aggregated descriptors
- Generate run-time test for the pruned tree

Minimal run-time overhead: $O(1)$
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>STATIC</td>
<td>Symbolic analysis</td>
<td>Symbolic analysis</td>
<td></td>
</tr>
<tr>
<td>DYNAMIC</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
Run-time Linear Memory Access Descriptor (RT_LMAD)

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

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

\[ S = \text{RT_LMAD} \]

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

\[ \text{LMAD} = \text{Start} + [\text{Stride1:Span1, Stride2:Span2, ...}] \]
Parallelism Detection: Hybrid 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 RT_LMADs
Compile Time Driver - Compiler Integration

Normalization filters: Multiple entries, premature loop exits, unstructured code

Other passes: Constant propagation, Induction variable recognition, Range dictionary, GSA

Hybrid analysis: Aggregation, parallelism detection, code generation

Backend passes: FORTRAN code generation
Sort Call Graph topologically

FOR EACH subprogram

Sort CDG topologically

FOR EACH statement

Process(statement)
Aggregation Across an Iteration Space

- WRITE pattern for a:
  
  $$1+[1:99]$$

- 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+[1:99]

• This case also solved at compile-time

SUBROUTINE Rfft(cc, ch)
INTEGER na, cc(*), ch(*)
ch(1:100) = d(1:100)
nana = 1
DO j=1,3
   na = 1 - na
   IF (na.EQ.0) THEN
      CALL Rad(cc, ch)
   ELSE
      CALL Rad(ch, cc)
   ENDIF
ENDDO
Gate Operator

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

- WRITE descriptor on \texttt{cc}
- Recurrence on \texttt{na} with no close form

\begin{verbatim}
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
ENDRfft
\end{verbatim}
SUBROUTINE Rfft(cc, ch)
INTEGER na, cccccc(*),(*),(*), chchch(:,*)

chchch(1:100) = chchch(1:100)
na = 1
DO j=1,3
    CALL Rfft((cccccc,, chchch))
    CALL Rfft((chchch, , , cccccc))
ENDDO

SUBROUTINE Run(SUBROUTINE Run(SUBROUTINE Run(www, , , ddd)))
INTEGER w(*), d(100,*)

DO j=1,1000
    CALL Rfft(w, d(1,j))
ENDDO

WF pattern on array w
na - local to Rfft
Detecting Loop Parallelism
Compile Time Decisions

- **dep_fa**
  - NO: CT Parallel
  - MAYBE: Unclassified Generate RT Test
  - YES: CT Sequential STOP

Generate parallel execution code \((dep_o, is_red)\)
Detecting Data Dependencies

- Loop expression: \( j = 1,N \)
- Per-iteration aggregated descriptors \( \text{RO}_j, \text{WF}_j, \text{RW}_j \)
- \( \text{RO} \bowtie \text{WF} \):

  Similar for \( \text{RW} \bowtie \text{WF}, \text{RO} \bowtie \text{RW}, \text{RW} \) overlaps

- Similar for output dependencies
• Possible dependence on array a
• Reduction on scalar sum
• HTML report

```
DO j=1, n
    a(j) = a(j+40) / 2
    sum = sum + a(j)
ENDDO
```
Code Generation

• Run-time analysis:
  – RT_LMAD → FORTRAN: attribute grammar
  – **Evaluate descriptors** at run-time, minimize overhead
    • Scalar comparisons
    • Inspector – executor
    • Speculative execution
  – Parallelism detection **decisions**
Run-time Complexity

PROGRAM main
   INTEGER w(100)
   INTEGER d(100,1000)
   step=0
   DO WHILE (step.LT.100)
      CALL Run()
      step=step+1
   ENDDO

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

SUBROUTINE Rfft(cc, ch)
   INTEGER cc(*), ch(*), na
   na=1
   DO j=1,3
      a(j)=2*b(j)+1
   ENDDO

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

PROGRAM main
   INTEGER www(*), (*), (*), ddd(100,1000)
   step=0
   DO WHILE (step.LT.100)
      CALL Run()
      step=step+1
   ENDDO

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

SUBROUTINE SUBROUTINE SUBROUTINE RfftRfftRfft(((cccccc,,, chchch)))
   INTEGER INTEGER INTEGER ccccc(*), (*), (*), chchch(*), (*), (*), nanana
   nanana=1
   DO j=1,3
      nanana=1---nanana
      IF (IF (IF (nanana.EQ.0) THEN.EQ.0) THEN.EQ.0) THEN
         CALL Rad(cc, ch)
      ELSE
         CALL RadRadRad(((chchch, , , ccccc)))
      ENDIF
   ENDDO

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

SUBROUTINE SUBROUTINE SUBROUTINE RfftRfftRfft(((cccccc,,, chchch)))
   INTEGER INTEGER INTEGER ccccc(*), (*), (*), chchch(*), (*), (*), nanana
   nanana=1
   DO j=1,3
      nanana=1
      IF (na.EQ.0) THEN
         CALL Rad(cc, ch)
      ELSE
         CALL RadRadRad(((chchch, , , ccccc)))
      ENDIF
   ENDDO
Reference-by-reference pure RT test

Hybrid Analysis

Number of operations necessary for computing access pattern

\[ 100 \times 1000 \times 3 \times 100 = 30,000,000 \]
Reducing Complexity

- At compile-time
  - Loop invariant RT_LMAD hoisting
  - Logic inference
  - Set identities, e.g. \((A-B)-A=\emptyset\)
  - Lattice identities, e.g. \(A-T=\emptyset\)

- At run-time
  - Contiguous aggregation
Compilation Complexity

• Memory usage: $O(N)$
  – Threshold for compile-time aggregation

• Worst-case running time: $O(N^2)$
  – Never reached in practice

$N =$ number of static data references
# Experimental Results

**PERFECT suite**

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Parallel coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADM</td>
<td>99.63%</td>
</tr>
<tr>
<td>DYFESM</td>
<td>99.72%</td>
</tr>
<tr>
<td>MDG</td>
<td>99.87%</td>
</tr>
<tr>
<td>TRACK</td>
<td>98.00%</td>
</tr>
</tbody>
</table>

**16 processor HP V 2200**

<table>
<thead>
<tr>
<th>Hand Parallel</th>
<th>Dynamic Analysis</th>
</tr>
</thead>
<tbody>
<tr>
<td>Static Analysis</td>
<td>Hybrid Analysis</td>
</tr>
</tbody>
</table>
Speedup for ADM

- Hand Parallel
- Static Analysis
- Hybrid Analysis
- Run-time (LRPD)

Number of Processors vs. Speedup
Speedup for DYFESM

- Hand Parallel
- Static Analysis
- Hybrid Analysis
- Run-time (LRPD)
Speedup for MDG

Number of Processors

- Hand Parallel
- Static Analysis
- Hybrid Analysis
- Run-time (LRPD)
Speedup for TRACK

Number of Processors

Speedup

Hand Parallel
Static Analysis
Hybrid Analysis
Run-time (R-LRPD)
Conclusions

• **Bridges** static and dynamic analyses
  – Benefits: same as fully dynamic methods
  – Costs: reduced by aggregation factors

• **Application**: automatic parallelization
  – Practically full coverage
  – Very low run-time overhead

• **Future applications** of Hybrid Analysis
  – Input sensitivity
  – Automatic checkpointing