Array privatization in IBM static compilers

-- technical report

CASCON 2005
Authors

Guansong Zhang, Erik Charlebois
and Roch Archambault

Compiler Development Team
IBM Toronto Lab, Canada
Overview

- Introduction and motivation
- Array data flow analysis
- Array data privatization
- Performance results
- Future work
- Possible usage
Expose limitations

- Compare SPEC2000FP and SPECCOMP
- SPECCOMP achieves good performance and scalability
  - Compare between explicit and auto-parallelization
- Expose missed opportunities
- 10 common benchmarks
  - Compare on a loop-to-loop basis
Improved auto-parallelization performance

One CPU vs. two CPU runs

Sequential
Parallel
Parallel+manual

time (sec.)

wupwise  swim  mgrid  applu  galgel  facerec  apsi  lucas  fma3d  sixtrack  mesa  art  equake  ammp
Array privatization example

a = 5

do 40 j = 1, n

   do 20 i = 2, m
      a(i) = b(i) + c(i)
   20    continue

   do 30 i = 2, m
      a(i) = a(i-1) + 4
   30    continue

40    continue

print *, a(2:m)
begin
for each loop nest in a procedure do
  for each loop in the nest in the depth-first order (outer first) do
    if the loop is user parallel then
      break
    if the loop is marked sequential, has side-effects etc then
      continue
    if the loop has loop carried dependence then
      try splitting the loop to eliminate dependence
      if dependence not eliminated then
        continue
    if loop cost is known at compile time then
      if the loop has not enough cost then
        break
    else
      Insert code for run-time cost estimate
  Mark this loop auto parallel
break
Pre-parallelization Phase

- Induction variable identification
- Scalar Privatization --- only scalar!
- Reduction finding
- Loop transformations favoring parallelism
The concept of data privatization

- Data is local to each loop iteration
  
  ```
  Do I = 1, 10
      Temp = ...
  ... = ... Temp ...
  ... = ... Temp ...
  Enddo
  ```

- Purpose: eliminating loop carried dependences.
The concept of data privatization (cont.)

- **Array as temp data**

```fortran
  do J = 1, 10
    do I = 1, 10
      Temp(I) = ...
    end do
  end do

  do I = 1, 10
    ... = ... Temp(I) ...
  enddo

  enddo
```
Array data flow and its structure

- Similar to data flow
  - MayDef: array elements that may be written.
  - MustDef: array elements that are definitely written.
  - UpExpUse: array elements that may have an upward exposed use
    - a use not preceded by a definition along a path from the loop header
  - LiveOnExit: array elements that are used after the loop region.

- GARs: Guarded Array Regions (GARs).
  - A GAR is a tuple(G,D),
    - D is a bounded Regular Section Descriptor (RSD) for the accessed array section,
    - G is a guard that specifies the condition under which D is accessed

- Notes: many papers discussed the issue
Array privatization algorithm

for each array A in the loop do
  for each GAR of A in the MayDef do
    if (GAR in all iterations intersects the UpExpUse of A) then
      Give up privatizing A
    else
      if (GAR intersects the LiveOnExit of A) then
        if (MustDef of A contains MayDef of A in all iterations) then
          Mark GAR in MayDef as private
        else
          Mark GAR in MustDef as last private
      else
        if (MustDef of A contains LiveOnExit of A) then
          Mark GAR in MayDef as private
        else
          Mark GAR in LiveOnExit as last private
      else
        Mark GAR in MayDef as private
  if (UpExpUse of A exist) then
    Mark GAR in UpExpUse as first private
Loop normalization and array data flow

- Normalized loop

```c
if (gard-expression) goto gard_label
prelog ...
    init induction variable to lower bound
loop_label:
    loop body ...
    computation based on induction variable
latch ...
    increase induction variable
    if (induction variable < upper bound)
        goto loop_label
epilog ...
    restore values if needed
gard_label:
    outside the loop
```
Alias analysis and array data flow

- Ideal situation: no alias at all.
  - Otherwise, you can not tell what is the precise intersection of the two array section involved

- Alias coming from:
  - Structural members, e.g. scalar replacement
  - Function parameters,
    - array is a shadow (not mapped data, alias to any global array)

- Procedure summary may help
  - Alias as fall back
Possible parallelization results

\[ a = 5 \]

\[
\text{!omp parallel do private a} \\
\text{!omp firstprivate a, lastprivate a} \\
\text{do 40 j = 1, n} \\
\text{do 20 i = 2, m} \\
\text{a(i) = b(i) + c(i)} \\
\text{20 continue} \\
\text{do 30 i = 2, m} \\
\text{a(i) = a(i-1) + 4} \\
\text{30 continue} \\
\text{40 continue} \\
\text{print *, a(2:m)}
\]
do i3=2,n3-1 ! This Loop cannot be automatically parallelized.
! A dependency is carried by variable "u1".
! U1 and U2 are local temporary variables, so that
! the loop should be parallelized
  do i2=2,n2-1
    do i1=1,n1
      ! Loop is parallelized
      u1(i1) = u(i1,i2-1,i3) + u(i1,i2+1,i3) + u(i1,i2,i3-1) + u(i1,i2,i3+1)
      u2(i1) = u(i1,i2-1,i3-1) + u(i1,i2+1,i3-1) + u(i1,i2-1,i3+1) + u(i1,i2+1,i3+1)
    enddo
    do i1=2,n1-1 ! Loop is parallelized
      r(i1,i2,i3) = v(i1,i2,i3) - a(0) * u(i1,i2,i3) - a(2) * ( u2(i1) + u1(i1-1) + u1(i1+1) ) - a(3) * ( u2(i1-1) + u2(i1+1) )
    enddo
  enddo
endo
NAS MG (-O3 –qhot –q64)
Summary

- Challenges
  - Compilation time
    - Work with other optimizations
      - Loop unroll
    - Graph complexity
      - Number of branches
    - Array section calculation accuracy
  - Memory usage
    - Managing and reusing
Summary (cont.)

- Further improvement:
  - Inter-procedural array data flow
    - Procedure summary
    - More accurate section information instead of using alias
  - Symbolic range analysis
    - Expression simplifier: lot of room to be improved
  - Compilation efficiency

- Possible usage
  - Auto parallelization
  - Array contraction
  - Array coalescing