Techniques for Reducing the Overhead of Run-time Parallelization

Hao Yu and Lawrence Rauchwerger
Department of Computer Science
Texas A&M University
rwerger@cs.tamu.edu
Reduce Run-time Overhead

What is the Problem?

• Parallel programming is difficult.
• Run-time technique can succeed where static compilation fails when access patterns are input dependent.

Goal of present work:

• Run-time overhead should be reduced as much as possible.
• Different run-time techniques should be explored for sparse code.
Fundamental Work: LRPD test

Main Idea:
• Speculatively apply the most promising transformations.
• Speculatively execute the loop in parallel and subsequently test the correctness of the execution.
• Memory references are marked in loop.
• If the execution was incorrect, the loop is re-executed sequentially.

Related Ideas:
• Array privatization, reduction parallelization is applied and verified.
• When using processor-wise test, cross-processor dependence are tested.
LRPD Test: Algorithm

Main components for speculative execution:

• Checkpointing/restoration mechanism
• Error(hazard) detection method for testing the validity of the speculative parallel execution
• Automatable strategy

- Initialize shadow arrays
- Checkpoint shared arrays
- Execute loop as DOALL
- Dependence analysis
  - Success?
    - Yes: Commit result
    - No: Restore from checkpoint
- Re-execute loop in serial
- Done
**LRPD Test: Example**

```plaintext
C C 0 0 : R e d u c i n g  R u n - t i m e  O v e r h e a d 5

L R P D  T e s t:  E x a m p l e

O r i g i n a l  l o o p  a n d  i n p u t  d a t a

L o o p  t r a n s f o r m e d  f o r  s p e c u l a t i v e  e x e c u t i o n .  T h e
m a r k w r i t e  a n d  m a r k r e a d  o p e r a t i o n s  u p d a t e  t h e
a p p r o p r i a t e  s h a d o w  a r r a y .

A n a l y s i s  o f  s h a d o w  a r r a y s  a f t e r  l o o p  e x e c u t i o n .

---

do i=1,5
   z = A(K(i))
   if (B(i).eq..true.) then
      A(L(i)) = z + C(i)
   endif
endo

B(1:5) = (1 0 1 0 1)
K(1:5) = (1 2 3 4 1)
L(1:5) = (2 2 4 4 2)

do i=1,5
   markread(K(i))
   z = A(K(i))
   if (B(i).eq..true.) then
      markwrite(L(i))
      A(L(i)) = z + C(i)
   endif
endo

d o  i=1,5
   z = A(K(i))
   if (B(i).eq..true.) then
      A(L(i)) = z + C(i)
   endif
endo

---

<table>
<thead>
<tr>
<th>Operation</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1 2 3 4 5</td>
</tr>
<tr>
<td>Aw</td>
<td>0 1 0 1 0</td>
</tr>
<tr>
<td>Ar</td>
<td>1 1 1 1 0</td>
</tr>
<tr>
<td>Anp</td>
<td>1 1 1 1 0</td>
</tr>
<tr>
<td>Aw(:) ^ Ar(:)</td>
<td>0 1 0 1 0</td>
</tr>
<tr>
<td>Aw(:) ^ Anp(:)</td>
<td>0 1 0 1 0</td>
</tr>
<tr>
<td>Atw</td>
<td>3</td>
</tr>
<tr>
<td>Atm</td>
<td>2</td>
</tr>
</tbody>
</table>

---

Original loop and input data

Loop transformed for speculative execution. The `markwrite` and `markread` operations update the appropriate shadow array.

Analysis of shadow arrays after loop execution.

CC00: Reducing Run-time Overhead
Sparse Applications

- The dimension of the array under test is much larger than the number of distinct elements referenced by loop.
- Use of shadow arrays is very expensive.
- Need compacted shadow structure.
- Rely on indirect, multi-level addressing.
- Example: SPICE 2G6.
Overhead minimization

Overhead Reduced

- The number of markings for memory references during speculative execution.
- Speculate about the data structures and reference patterns of the loop and customize the shape and size of shadow structure.
Redundant marking elimination

**Same address type based aggregation**

- Memory references are classified as: RO, WF, RW, NO
- To mark minimal sites by using dominance relationship.
- Algorithm relies on recursive aggregation by DFS traversal on CDG.
- Combine markings based on *rules*.
- Boolean expression simplification will enhance the ability of reducing more redundant markings.
- Loop invariant predicates of references will enable extracting a inspector loop.
Redundant marking elimination: Rules

A

\[ \text{NO} \quad \text{RO} \quad \text{NO} \quad \text{RO} \quad \text{NO} \quad \text{NO} \]

F

\[ \text{RO} \quad \text{WF} \quad \text{WF} \quad \text{RW} \quad \text{NO} \quad \text{NO} \]
Grouping of related references

• Two references are related if satisfy:
  1) The subscripts of two references are of form \( ptr + \text{affine function} \)
  2) The corresponding \( ptrs \) are the same.

• Related references of the same type can be grouped for the purpose of marking reduction.

• Grouping procedure is to find a minimum number of disjoint sets of references.

• Result:
  1) reduced number of marking instruction.
  2) reduced size of shadow structure.
CDG and colorCDG

- colorCDG represents the predicates clearer than CDG does.
- colorCDG and CDG contain the same information in a clearer form.
Grouping Algorithm

Procedure: Grouping
Input:
    Statement N;
    Predicate C;
Output:
    Groups return_groups;
Local:
    Groups branch_groups;
    Branch branch;
    Statement node;
    Predicate path_condition;

```
return_groups = CompGroups (N, C)
if (N.succ_entries > 0)
    for (branch in N.succ)
        Initialize(branch_groups)
        path_condition = C & branch.predicate
        for (node in B.succ)
            branch_groups ∪= Grouping(node, path_condition)
        end
    end
    return_groups ∩= branch_groups
end
return return_groups
```
Sparse access classification

- Identify base-pointers
- Classify references of same base-pointers to:

**A)** Monotonic Access + constant stride

```
1 → 2 → 3 → 4
```

(1, 4, 4)

**B)** Monotonic Access + variable stride

```
1 → 2 → 3 → 4
```

Array + Range Info

**C)** Random access

```
1 → 4 → 2, 3
```

Hash Table + Range Info
Dependence Analysis

- Run-time marking routines are adaptive.
- Type of references is recorded in bit vector.
- Access and analysis of the shadow structures are expensive normally.
- It can be cheap when speculation on the access pattern is correct.

CC00: Reducing Run-time Overhead
Experimental Results

Results

• Marking overhead reduction for speculative loop.
• Shadow structures for sparse codes:
  SPICE 2G6, BJT loop.

Experimental Setup

• Experiments run on:
  16 processor HP-V class, 4GB memory, HPUX11 operation system.
• Techniques are implemented in Polaris infrastructure.
• Codes: SPICE 2G6, P3M, OCEAN, TFFT2
Marking Overhead Reduction

• Polaris + Run-time pass + Grouping

• Speedup Ratio = \( \frac{\text{Exec. time of loop (without grouping)}}{\text{Exec. time of loop (with grouping)}} \)

<table>
<thead>
<tr>
<th>Program: Loop</th>
<th>Marking point reduction %</th>
<th>Speedup Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Static</td>
<td>Dynamic</td>
</tr>
<tr>
<td>SPICE 2G6: BJT</td>
<td>91.3 %</td>
<td>83.88 %</td>
</tr>
<tr>
<td>P3M: PP_do100</td>
<td>50 %</td>
<td>40.57 %</td>
</tr>
<tr>
<td>OCEAN: Ftrvmt_do9109</td>
<td>50 %</td>
<td>50 %</td>
</tr>
<tr>
<td>TFFT2: Cfftz_do#1</td>
<td>55 %</td>
<td>68.75 %</td>
</tr>
</tbody>
</table>
Experiment: SPICE 2G6, BJT loop

SPICE 2G6, BJT loop:
• BJT is about 30% of total sequential execution Time.
• Unstructured Loop ⇒ DO loop
• Distribute the DO loop to
  – Sequential loop collects all nodes in the linked list into a temporary array.
  – Parallel loop does the real work.
• Most indices are based on common base-pointer.
• It does its own memory management.

We applied:
• Grouping method.
• Choice of shadow structures.
Experiment: Speedup

Input: Extended from long input of Perfect suit

Input: Extended from 8-bits adder
Experiment: Speedup

Input: Extended from long input of SPEC 92

Input: Extended from 8-bits adder
Experiment: Execution Time

Input: Extended from long input of Perfect suit

Input: Extended from 8-bits adder
Conclusion

Proposed Techniques
- increased potential speedup, efficiency of run-time parallelized loops.
- will dramatically increase coverage of automatic parallelism of Fortran programs.

Techniques used for SPICE may be applicable to C code.