Advanced Compiler Construction Theory And Practice

Introduction to loop dependence and Optimizations

A little about myself

Qing Yi

- *Ph.D.* Rice University, USA.
- Associate Professor, University of Colorado at Colorado Springs

Research Interests

- Compiler construction, software productivity
- program analysis and optimization for high-performance computing
- Parallel programming

Overall goal: develop tools to improve the productivity and efficiency of programming

- Optimizing compilers for high performance computing
- Programmable optimization and tuning of Scientific codes

General Information

- Reference books
  - Optimizing Compilers for Modern Architectures: A Dependence-based Approach,
    - Randy Allen and Ken Kennedy, Morgan-Kauffman Publishers Inc.
  - Book Chapter: Optimizing And Tuning Scientific Codes
    - Qing Yi. SCALABLE COMPUTING AND COMMUNICATIONS:THEORY AND PRACTICE.
    - http://www.cs.uccs.edu/~qyi/papers/BookChapter11.pdf
  - Structured Parallel Programming Patterns for Efficient Computation
    - Michael McCool, James Reinders, and Arch Robison. Morgan Kaufmann. 2012.
    - http://parallelbook.com/sites/parallelbook.com/files/SC13_20131117_Intel_McCool_Robison_Reinders_Hebenstreit.pdf
- Course materials
  - http://www.cs.uccs.edu/~qyi/classes/Dragonstar
- The POET project web site:
  - http://www.cs.uccs.edu/~qyi/poet/index.php

High Performance Computing

- Applications must efficiently manage architectural components
  - Parallel processing
    - Multiple execution units --- pipelined
    - Vector operations, multi-core
    - Multi-tasking, multi-threading (SIMD), and message-passing
  - Data management
    - Registers
    - Cache hierarchy
    - Shared and distributed memory
  - Combinations of the above
- What are the compilation challenges?

Optimizing For High Performance

- Goal: eliminating inefficiencies in programs
- Eliminating redundancy: if an operation has already been evaluated, don't do it again
  - Especially if the operation is inside loops or part of a recursive evaluation
  - All optimizing compilers apply redundancy elimination,
    - e.g., loop invariant code motion, value numbering, global RE
- Resource management: reorder operations and/or data to better map to the targeting machine
  - Reorder computation(operations)
    - parallelization, vectorization, pipelining, VLIW, memory reuse
    - Instruction scheduling and loop transformations
  - Re-organization of data
    - Register allocation, regrouping of arrays and data structures

Optimizing For Modern Architectures

- Key: reorder operations to better manage resources
  - Parallelization and vectorization
  - memory hierarchy management
  - Instruction and task/threads scheduling
  - Interprocedural (whole-program) optimizations

- Most compilers focus on optimizing loops, why?
  - This is where the application spends most of its computing time
  - What about recursive function/procedural calls?
    - Extremely important, but often left unoptimized…

Compiler Technologies

- Source-level optimizations
  - Most architectural issues can be dealt with by restructuring the program source
    - Vectorization, parallelization, data locality enhancement
  - Challenges:
    - Determining when optimizations are legal
    - Selecting optimizations based on profitability
- Assembly level optimizations
  - Some issues must be dealt with at a lower level
    - Prefetch insertion
    - Instruction scheduling
- All require some understanding of the ways that instructions and statements depend on one another (share data)

Syllabus

- Dependence Theory and Practice
  - Automatic detection of parallelism.
  - Types of dependences; Testing for dependence;
- Memory Hierarchy Management
  - Locality enhancement; data layout management;
  - Loop interchange, blocking, unroll&jam, unrolling,…
- Loop Parallelization
  - More loop optimizations: OMP parallelization, skewing, fusion, …
  - Private and reduction variables
- Pattern-driven optimization
  - Structured Parallelization Patterns
  - Pattern-driven composition of optimizations
- Programmable optimization and tuning
  - Using POET to write your own optimizations

Dependence-based Optimization

- **Bernstein's Conditions**
  - **it is safe to run two tasks R1 and R2 in parallel if none of the following holds:**
    - **R1 writes into a memory location that R2 reads**
    - **R2 writes into a memory location that R1 reads**
    - **Both R1 and R2 write to the same memory location**
- There is a dependence between two statements if
  - They might access the same location,
  - There is a path from one to the other, and
  - One of the accesses is a write
- Dependence can be used for
  - Automatic parallelization
  - Memory hierarchy management (registers and caches)
  - Scheduling of instructions and tasks

Dependence - Static Program Analysis

- Program analysis --- support software development and maintenance
  - Compilation --- identify errors without running program
    - Smart development environment (check simple errors as you type)
  - Optimization --- cannot not change program meaning
    - Improve performance, efficiency of resource utilization
    - Code revision/re-factoring ==> reusability, maintainability,
  - Program correctness --- Is the program safe/correct?
    - Program verification -- Is the implementation safe, secure?
    - Program integration --- are there any communication errors?
- In contrast, if the program needs to be run to figure out information, it is called dynamic program analysis.

Data Dependences

- **There is a data dependence from statement S_1 to S_2 if**
  1. **Both statements access the same memory location,**
  2. **At least one of them stores onto it, and**
  3. **There is a feasible run-time execution path from S_1 to S_2**
- Classification of data dependence
  - True dependences (Read After Write hazard)
    S_2 depends on S_1 is denoted by $S_1 \delta S_2$
  - Anti dependence (Write After Read hazard)
    S_2 depends on S_1 is denoted by $S_1 \delta^{-1} S_2$
  - Output dependence (Write After Write hazard)
    S_2 depends on S_1 is denoted by $S_1 \delta^0 S_2$
- Simple example of data dependence:

S_1 **PI = 3.14**

S_2 **R = 5.0**

S_3 **AREA = PI * R ** 2**

Transformations

- A reordering Transformation
  - Changes the execution order of the code, without adding or deleting any operations.
- Properties of Reordering Transformations
  - It does not eliminate dependences, but can change the ordering (relative source and sink) of a dependence
  - If a dependence is reverted by a reordering transformation, it may lead to incorrect behavior
- A reordering transformation is safe if it preserves the relative direction (i.e., the source and sink) of each dependence.

Dependence in Loops

```
   DO I = 1, N
S1  A(I+1) = A(I) + B(I)
   ENDDO
```

```
   DO I = 1, N
S1  A(I+2) = A(I) + B(I)
   ENDDO
```

- In both cases, statement S1 depends on itself
  - However, there is a significant difference
- We need to distinguish different iterations of loops
  - The iteration number of a loop is equal to the value of the loop index (loop induction variable)
  - Example:

```
    DO I = 0, 10, 2
S1        <some statement>
    ENDDO
```

- What about nested loops?
  - Need to consider the nesting level of a loop

Iteration Vectors

- Given a nest of n loops, iteration vector *i* is
  - A vector of integers $\{i_1, i_2, ..., i_n\}$ where i_k, $1 \le k \le n$ represents the iteration number for the loop at nesting level k
- Example:

```
DO I = 1, 2
   DO J = 1, 2
S1       <some statement>
   ENDDO
ENDDO
```

  - The iteration vector (2, 1) denotes the instance of S_1 executed during the 2nd iteration of the I loop and the 1st iteration of the J loop

Loop Iteration Space

- For each loop nest, its iteration space is
  - The set of all possible iteration vectors for a statement
  - Example:

```
DO I = 1, 2
  DO J = 1, 2
S1      <some statement>
  ENDDO
ENDDO
```

The iteration space for S1 is { (1,1),(1,2),(2,1),(2,2) }

Ordering of Iterations

- Within a single loop iteration space,
  - We can impose a lexicographic ordering among its iteration Vectors
- Iterations i precedes j, denoted i < j, iff
  - Iteration i is evaluated before j
  - That is, for some nesting level k

    1. i[i:k-1] < j[1:k-1], or
    2. i[1:k-1] = j[1:k-1] and $i_n < j_n$
  - Example: (1,1) < (1,2) < (2,1) < (2,2)

Loop Dependence

There exists a dependence from statement S1 to S2 in a common nest of loops if and only if

- there exist two iteration vectors i and j for the nest, such that
  - (1) i < j or i = j and there is a path from S1 to S2 in the body of the loop,
  - (2) statement S1 accesses memory location M on iteration i and statement S2 accesses location M on iteration j, and
  - (3) one of these accesses is a write.

Distance and Direction Vectors

- Consider a dependence in a loop nest of n loops
  - Statement S1 on iteration i is the source of dependence
  - Statement S2 on iteration j is the sink of dependence
- The distance vector is a vector of length n d(i,j) such that: $d(i,j)_k = j_k - I_k$
- The direction Vector is a vector of length n D(i,j) such that (Definition 2.10 in the book)

$$D(i,j)_k = \begin{cases} \text{"<" if } d(i,j)_k > 0 \\ \text{"=" if } d(i,j)_k = 0 \\ \text{">" if } d(i,j)_k < 0 \end{cases}$$

- What is the dependence distance/direction vector?

```
DO I = 1, N
  DO J = 1, M
    DO K = 1, L
S1     A(I+1, J, K-1) = A(I, J, K) + 10
```

Loop-carried and Loop-independent Dependences

- If in a loop statement S2 on iteration ***j*** depends on S1 on iteration ***i***, the dependence is
  - Loop-carried if either of the following conditions is satisfied
    - S1 and S2 execute on different iterations i.e., $i \neq j$
    - ***d***(***i,j***) > **0** i.e. ***D***(***i,j***) contains a "<" as leftmost non "=" component
  - Loop-independent if either of the conditions is satisfied
    - S1 and S2 execute on the same iteration i.e., ***i=j***
    - ***d***(***i,j***) = **0,** i.e. ***D***(***i,j***) contains only "=" component
    - NOTE: there must be a path from S1 to S2 in the same iteration
- Example:

```
DO I = 1, N
S1      A(I+1) = F(I)+ A(I)
S2      F(I) = A(I+1)
ENDDO
```

Level of loop dependence

- The level of a loop-carried dependence is the index of the leftmost non-“=” of D(i,j)
  - A level-k dependence from S_1 to S_2 is denoted $S_1\ \delta_k\ S_2$
  - A loop independent dependence from S1 to S_2 is denoted $S_1\delta_\infty S_2$
- Example:

```
DO I = 1, 10
   DO J = 1, 10
      DO K = 1, 10
S1       A(I, J, K+1) = A(I, J, K)
S2        F(I,J,K) = A(I,J,K+1)
      ENDDO
   ENDDO
ENDDO
```

Loop Optimizations

- A loop optimization may
  - Change the order of iterating the iteration space of each statement
  - Without altering the direction of any original dependence
- Example loop transformations
  - Change the nesting order of loops
  - Fuse multiple loops into one or split one into multiple
  - Change the enumeration order of each loop
  - And more …
- Any loop reordering transformation that

  (1) does not alter the relative nesting order of loops and

  (2) preserves the iteration order of the level-k loop

  preserves all level-k dependences.

Dependence Testing

```
DO i1 = L1, U1, S1
     DO i2 = L2, U2, S2
        ...
          DO in = Ln, Un, Sn
     S1     A(f1(i1,...,in),...,fm(i1,...,in)) = ...
     S2    ... = A(g1(i1,...,in),...,gm(i1,...,in))
          ENDDO
       ...
     ENDDO
ENDDO
```

- A dependence exists from S1 to S2 iff there exist iteration vectors x=(x1,x2,…,xn) and y=(y1,y2,…,yn) such that

  (1) x is lexicographically less than or equal to y;

  (2) the system of ***diophatine equations*** has an integer solution:

  fi(x) = gi(y) for all $1 \le i \le m$

  i.e. fi(x1,…,xn)-gi(y1,…,yn)=0 for all $1 \le i \le m$

Example

```
DO I = 1, 10
   DO J = 1, 10
      DO K = 1, 10
S1       A(I, J, K+1) = A(I, J, K)
S2       F(I,J,K) = A(I,J,K+1)
      ENDDO
   ENDDO
ENDDO
```

- To determine the dependence between A(I,J,K+1) at iteration vector (I1,J1,K1) and A(I,J,K) at iteration vector (I2,J2,K2), solve the system of equations

  I1 = I2; J1=J2; K1+1=K2; 1 <= I1,I2,J1,J2,K1,K2 <= 10

  - Distance vector is (I2-I1,J2-J1,K2-K1)=(0,0,1)
  - Direction vector is (=,=,<)
  - The dependence is from A(I,J,K+1) to A(I,J,K) and is a true dependence

The Delta Notation

- Goal: compute iteration distance between the source and sink of a dependence

```
DO I = 1, N
   A(I + 1) = A(I) + B
ENDDO
```

  - Iteration at source/sink denoted by: $I0$ and $I0+\Delta I$
  - Forming an equality gets us: $I0 + 1 = I0 + \Delta I$
  - Solving this gives us: $\Delta I = 1$
- If a loop index does not appear, its distance is *
  - * means the union of all three directions <,>,=

```
DO I = 1, 100
DO J = 1, 100
   A(I+1) = A(I) + B(J)
```

  - The direction vector for the dependence is (<, *)

Complexity of Testing

- Find integer solutions to a system of Diophantine Equations is NP-Complete
  - Most methods consider only linear subscript expressions
- Conservative Testing
  - Try to prove absence of solutions for the dependence equations
  - Conservative, but never incorrect
- Categorizing subscript testing equations
  - ZIV if it contains no loop index variable
  - SIV if it contains only one loop index variable
  - MIV if it contains more than one loop index variables

    A(5,I+1,j) = A(1,I,k) + C

    5=1 is ZIV; I1+1=I2 is SIV; J1=K2 is MIV

Summary

- Introducing data dependence
  - What is the meaning of S2 depends on S1?
  - What is the meaning of $S_1 \delta S_2$, $S_1 \delta^{-1} S_2$, $S_1 \delta^0 S_2$?
  - What is the safety constraint of reordering transformations?
- Loop dependence
  - What is the meaning of iteration vector (3,5,7)?
  - What is the iteration space of a loop nest?
  - What is the meaning of iteration vector $I < J$?
  - What is the distance/direction vector of a loop dependence?
  - What is the relation between dependence distance and direction?
  - What is the safety constraint of loop reordering transformations?
- Level of loop dependence and transformations
  - What is the meaning of loop carried/independent dependences?
  - What is the level of a loop dependence or loop transformation?
  - What is the safety constraint of loop parallelization?
- Dependence testing theory