Recall Loop Fusion

Idea

– Combine multiple loop nests into one

Example

```
do i = 1,n
    A(i) = A(i-1)
enddo
do j = 1,n
    B(j) = A(j)/2
enddo
```

→

```
do i = 1,n
    A(i) = A(i-1)
    B(i) = A(i)/2
enddo
```

How?

Recall Legality Requirement

Dependences must be preserved

– *e.g.*, Flow dependences must not become anti dependences

```
do i = 1,n
    body1
enddo
do i = 1,n
    body2
enddo
```

All cross-loop dependences flow from body1 to body2

→

```
do i = 1,n
    body1
    body2
enddo
```

Ensure that fusion does not introduce dependences from body2 to body1

Loop Fusion Example

What are the dependences?

```
    do i = 1,n
s1      A(i) = B(i) + 1
    enddo

    do i = 1,n
s2      C(i) = A(i)/2
    enddo

    do i = 1,n
s3      D(i) = 1/C(i+1)
    enddo
```

$s_1\delta^f s_2$

$s_2\delta^f s_3$

What are the dependences?

```
    do i = 1,n
s1      A(i) = B(i) + 1
s2      C(i) = A(i)/2
s3      D(i) = 1/C(i+1)
    enddo
```

$s_1\delta^f s_2$

$s_3\delta^a s_2$

Fusion changes the dependence between s_2 and s_3, so fusion is illegal

May 4, 2015 Loop Transformations 3

Recall Loop Fission (Loop Distribution)

Idea

– Split a loop nest into multiple loop nests (the inverse of fusion)

Example

```
do i = 1,n
    A(i) = B(i) + 1
    C(i) = A(i)/2
enddo
```

```
do i = 1,n
    A(i) = B(i) + 1
enddo

do i = 1,n
    C(i) = A(i)/2
enddo
```

May 4, 2015 Loop Transformations 4

Loop Fission (cont)

Legality

– Fission is legal when the loop body contains no cycles in the dependence graph

```
do i = 1,n
   body1
   body2
enddo
```

```
do i = 1,n
   body1
enddo
do i = 1,n
   body2
enddo
```

Cycles cannot be preserved because after fission all cross-loop dependences flow from body1 to body2

Loop Fission Example

Recall our fusion example

```
   do i = 1,n
s1    A(i) = B(i) + 1
   enddo

   do i = 1,n
s2    C(i) = A(i)/2
   enddo

   do i = 1,n
s3    D(i) = 1/C(i+1)
   enddo
```

$s_1 \delta^f s_2$

$s_2 \delta^f s_3$

Can we perform fission on this loop?

```
   do i = 1,n
s1    A(i) = B(i) + 1
s2    C(i) = A(i)/2
s3    D(i) = 1/C(i+1)
   enddo
```

$s_1 \delta^f s_2$

$s_3 \delta^a s_2$

Do we have a contradiction?

Loop Fission Example (cont)

If there are no cycles, we can reorder the loops with a topological sort

```
     do i = 1,n
s1       A(i) = B(i) + 1
     enddo

     do i = 1,n
s3       D(i) = 1/C(i+1)
     enddo

     do i = 1,n
s2       C(i) = A(i)2
     enddo
```

$s_1 \delta^f s_2$ $s_3 \delta^a s_2$

Fission is legal

```
     do i = 1,n
s1       A(i) = B(i) + 1
s2       C(i) = A(i)/2
s3       D(i) = 1/C(i+1)
     enddo
```

$s_1 \delta^f s_2$ $s_3 \delta^a s_2$

Loop Interchange

Idea

- Swap the order of two loops to increase parallelism, to improve spatial locality, or to enable other transformations
- Also known as **loop permutation**

Example

```
do i = 1,n
   do j = 1,n
      x = A(2,j)
   enddo
enddo
```

This access strides through a row of A

```
do j = 1,n
   do i = 1,n
      x = A(2,j)
   enddo
enddo
```

This code is invariant with respect to the inner loop, yielding better locality

(Assuming column-major order for Fortran)

Loop Interchange (cont)

Example

```
do i = 1,n
    do j = 1,n
        x = A(i,j)
    enddo
enddo
```

This array has stride n access

```
do j = 1,n
    do i = 1,n
        x = A(i,j)
    enddo
enddo
```

This array now has stride 1 access

Legality of Loop Interchange

Case analysis of the direction vectors

(=,=)

The dependence is loop independent, so it is unaffected by interchange

(=,<)

The dependence is carried by the j loop.
After interchange the dependence will be (<,=), so the dependence will still be carried by the j loop, so the dependence relations do not change.

(<,=)

The dependence is carried by the i loop.
After interchange the dependence will be (=,<), so the dependence will still be carried by the i loop, so the dependence relations do not change.

Legality of Loop Interchange (cont)

Case analysis of the direction vectors (cont)

(<,<)

The dependence distance is positive in both dimensions.

After interchange it will still be positive in both dimensions, so the dependence relations do not change.

(<,>)

The dependence is carried by the outer loop.

After interchange the dependence will be (>,<), which changes the dependences and results in an illegal direction vector, so interchange is illegal.

(>,*) (=,>)

Such direction vectors are not possible for the original loop.

Loop Interchange Example

Consider the (<,>) case

```
do i = 1,n
   do j = 1,n
      C(i,j) = C(i+1,j-1)
   enddo
enddo
```

```
do j = 1,n
   do i = 1,n
      C(i,j) = C(i+1,j-1)
   enddo
enddo
```

Before

(1,1) C(1,1) = C(2,0)

(1,2) C(1,2) = C(2,1)

. . . d = (<,>) δ^a

(2,1) C(2,1) = C(3,0)

After

(1,1) C(1,1) = C(2,0)

(1,2) C(2,1) = C(3,0)

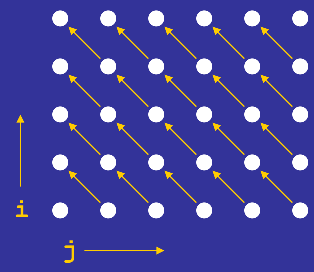
. . . d = (<,>) δ^f

(2,1) C(1,2) = C(2,1)

Example 2 from Last Lecture

Sample code

```
do i = 1,5
   do j = 2,6
      A(j,i) = A(j-1,i+1)+1
   enddo
enddo
```



Kind of dependence: Anti

Distance vector: (1, –1)

Fusion Exercise

Can we fuse these loop nests?

```
do i = 1,n
   X(i) = 0
enddo
do j = 1,n
   do k = 1,n
      X(k) = X(k)+A(k,j)*Y(j)
   enddo
enddo
```

δ^f

Fusion of these loops would violate this dependence

```
do i = 1,n
   X(i) = 0
   do k = 1,n
      X(k) = X(k)+A(k,i)*Y(i)
   enddo
enddo
```

Is there some transformation that will enable fusion of these loops?

Fusion Exercise (cont)

Use loop interchange to preserve dependences

```
do i = 1,n
   X(i) = 0
enddo
do k = 1,n
   do j = 1,n
      X(k) = X(k)+A(k,j)*Y(j)
   enddo
enddo
```

δ^f

⇒

```
do i = 1,n
   X(i) = 0
   do j = 1,n
      X(i) = X(i)+A(i,j)*Y(j)
   enddo
enddo
```

δ^f

May 4, 2015 Loop Transformations 15

Loop Unrolling

Motivation

- Reduces loop overhead
- Improves effectiveness of other transformations
  - Code scheduling
  - CSE

The Transformation

- Make n copies of the loop: n is the **unrolling factor**
- Adjust loop bounds accordingly

May 4, 2015 Loop Transformations 16

Loop Unrolling (cont)

Example

```
do i=1,n
   A(i) = B(i) + C(i)
enddo
```

⇒

```
do i=1,n by 2
   A(i)   = B(i) + C(i)
   A(i+1) = B(i+1) + C(i+1)
enddo
```

Details

- When is loop unrolling legal?
- Handle end cases with a cloned copy of the loop
  - Enter this special case if the remaining number of iteration is less than the unrolling factor

Loop Balance

Problem

- We'd like to produce loops with the right balance of memory operations and floating point operations
- The ideal balance is machine-dependent
  - *e.g.* How many load-store units are connected to the L1 cache?
  - *e.g.* How many functional units are provided?

Example

```
do j = 1,2*n
   do i = 1,m
      A(j) = A(j) + B(i)
   enddo
enddo
```

- The inner loop has 1 memory operation per iteration and 1 floating point operation per iteration
- If our target machine can only support 1 memory operation for every two floating point operations, this loop will be memory bound

What can we do?

Unroll and Jam

Idea

- Restructure loops so that loaded values are used many times per iteration

Unroll and Jam

- Unroll the outer loop some number of times
- Fuse (Jam) the resulting inner loops

Example

```
do j = 1,2*n
   do i = 1,m
      A(j) = A(j) + B(i)
   enddo
enddo
```

Unroll the Outer Loop

```
do j = 1,2*n by 2
   do i = 1,m
      A(j)   = A(j)   + B(i)
   enddo
   do i = 1,m
      A(j+1) = A(j+1) + B(i)
   enddo
enddo
```

May 4, 2015 Loop Transformations 19

Unroll and Jam Example (cont)

Unroll the Outer Loop

```
do j = 1,2*n by 2
   do i = 1,m
      A(j)   = A(j)   + B(i)
   enddo
   do i = 1,m
      A(j+1) = A(j+1) + B(i)
   enddo
enddo
```

Jam the inner loops

```
do j = 1,2*n by 2
   do i = 1,m
      A(j)   = A(j)   + B(i)
      A(j+1) = A(j+1) + B(i)
   enddo
enddo
```

- The inner loop has 1 load per iteration and 2 floating point operations per iteration
- We reuse the loaded value of **B(i)**
- The Loop Balance matches the machine balance

May 4, 2015 Loop Transformations 20

Unroll and Jam (cont)

Legality
- When is Unroll and Jam legal?

Disadvantages
- What limits the degree of unrolling?

Policies

Policies for improving locality and parallelism
- Many proposed ideas
- Few unified frameworks
- Locality framework [Wolf and Lam 1991]
  - Phrase a class of loop transformations as unimodular matrix transformations

Concepts

Using direction and distance vectors

Transformations

- What is the benefit?
- What do they enable?
- When are they legal?

Next Time

Lecture

- Dynamic compilation

Question of the Day

Q: Is it really a small world?

How many hops does it take to connect two random persons in the US?

A: According to an old study (more than 45 years old), only 3!
(Interestingly, many of the critical links were butchers.)

Q: How many hops does it take to connect two random persons in the world?

May 4, 2015 Loop Transformations 25

Prelude

A: 5, or 6 degrees separation

May 4, 201

Dynamic Compilation

Last time
- Loop transformations and parallelism

This time
- Dynamic compilation

Motivation

Limitations of static analysis
- Programs can have values and invariants that are known at runtime but unknown at compile time. Static compilers cannot exploit such values or invariants
- Many of the motivations for profile-guided optimizations apply here

Basic idea
- Perform translation at runtime when more information is known
- Traditionally, two types of translations are done
  - Runtime code generation
  - Partial evaluation

Partial Evaluation

Basic idea

- Take a general program and partially evaluate it, producing a specialized program that's more efficient

  e.g., f(a,b,c)→f'(a,b), where the result has its third parameter hard-coded into the implementation. f' is typically more efficient than f

- Exploit **runtime constants,** which are variables whose value does not change during program execution, *e.g.*, write-once variables

Exploiting runtime constants

- Perform constant propagation
- Eliminate memory ops
- Remove branches
- Unroll loops

Improves performance by moving computation from runtime to compile time

Programs with Runtime Constants

Interpreters: The program being interpreted is runtime constant

Simulators: The subject of simulation (circuit, cache, network) is runtime constant

Graphics renderers: The scene to render is runtime constant

Scientific simulations: Matrices can be runtime constants

Extensible OS kernels: Extensions to the kernel can be runtime constant

Examples

- A cache simulator might take the **line size** as a parameter
- A partially evaluated simulator might produce a faster simulator for the special case where the **line size** is 16

Partial Evaluation (cont)

Interesting theoretical results

- Can partially evaluate an interpreter with respect to a program (*i.e.*, compile it) [1st Futamura projection, 1971]
- Can partially evaluate a partial evaluator with respect to an interpreter (*i.e*, generate a compiler) [2nd Futamura projection, 1983]
- Can partially evaluate a partial evaluator with respect to a partial evaluator (i.e., generate a compiler generator) [3rd Futamura projection]

Early PE research focused on functional languages

- Recent work has moved to languages such as C and Java [Cook '11]

Key issue

- When do we stop partially evaluating the code when there is iteration or recursion?

May 4, 2015 Loop Transformations 31

Dynamic Compilation with DyC

DyC [Auslander, *et al* 1996]

- Apply ideas of Partial Evaluation
- Perform some of the Partial Evaluation at runtime
  - Can handle more runtime constants than Partial Evaluation
- Reminiscent of link-time register allocation in the sense that the compilation is performed in stages

Tradeoffs

- Must overcome the run-time cost of the dynamic compiler
  - Fast dynamic compilation: low overhead
  - High quality dynamically generated code: high benefit
- Ideal: dynamically translate code once, execute this code many times
- Implication: don't dynamically translate everything
  - Only perform dynamic translation where it will be profitable

May 4, 2015 Loop Transformations 32

Applying Dynamic Compilation

How do we know what will be profitable?

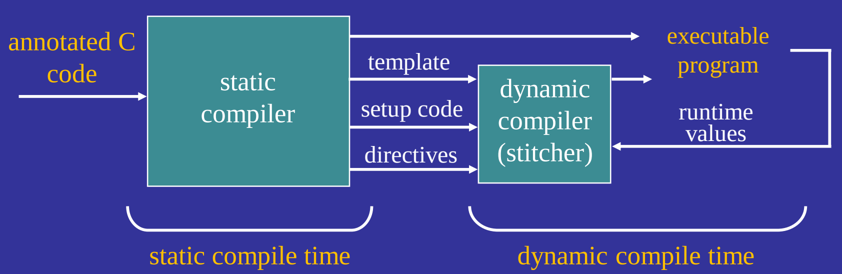
- Let user annotations guide the dynamic compilation process

System design

- Dynamic compilation for the C language
- Declarative annotations:
  - Identify pieces of code to dynamically compile: dynamic regions
  - Identify source code variables that will be constant during the execution of dynamic regions: runtime constants

May 4, 2015 Loop Transformations 33

Staged Compilation in DyC



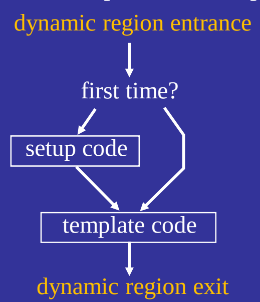
- Make the static compiler do as much work as possible
- Give the dynamic compiler as little work as possible

May 4, 2015 Loop Transformations 34

Dynamically Compiled Code

Static compiler

- Produces machine code **templates**, in addition to normal machine code
- Templates contain **holes** that will be filled with runtime constant values
- Generates **setup code** to compute the values of these runtime constants
- Together, the template and setup code will replace the original dynamic region



The Dynamic Compiler

The Stitcher

- Follows **directives**, which are produced by the static compiler, to copy code templates and to fill in holes with appropriate constants
- The resulting code becomes part of the executable code and is hopefully executed many times

The Annotations

```
cacheResult cacheLookup (void *addr, Cache *cache) {
   dynamicRegion(cache) { /* cache is a runtime constant */
      int blockSize = cache->blockSize;
      int numLines = cache->numLines;
      int tag = addr / (blockSize * numLines);
      int line = (add / blockSize) % numLines;
      setStructure **setArray = cache->lines[line]->sets;
      int assoc = cache->associativity;
      int set;

      unrolled for (set=0; set<assoc; set++) {
          if (setArray[set]dynamic->tag == tag)
              return CacheHit;
      }
      return CacheMiss;
   } /* end of dynamic region */
}
```


The Annotations

```
cacheResult cacheLookup (void *addr, Cache *cache) {
   dynamicRegion(cache) { /* cache is a runtime constant */
```

dynamicRegion(cache)

- Identifies a block that will be dynamically compiled
- Its arguments are runtime constants within the scope of the dynamic region
- The static compiler will compute additional runtime constants that are derived from this initial set

```
      }
      return CacheMiss;
   } /* end of dynamic region */
```

The Annotations

```
cacheResult cacheLookup (void *addr, Cache *cache) {
```

dynamic

- Any type of data can be considered constant
- In particular, contents of arrays and pointer-based structures are assumed to be runtime constant whenever they are accessed by runtime constant pointers
- To ensure that this assumption is correct, users must insert the **dynamic** annotation to mark pointer refs that are **not** constant

```
            if (setArray[set]dynamic->tag == tag)
                return CacheHit;
        }
        return CacheMiss;
    } /* end of dynamic region */
}
```

May 4, 2015 Loop Transformations 39

The Annotations

unrolled

- Directs the compiler to completely unroll a loop
- Loop termination must be governed by runtime constants
  - The static compiler can check whether this annotation is legal
- Complete unrolling is a critical optimization
  - Allows induction variables to become runtime constants

```
        unrolled for (set=0; set<assoc; set++) {
            if (setArray[set]dynamic->tag == tag)
                return CacheHit;
        }
        return CacheMiss;
    } /* end of dynamic region */
}
```

May 4, 2015 Loop Transformations 40

The Annotations

```
cacheResult cacheLookup (void *addr, Cache *cache) {
   dynamicRegion key(cache, foo) {
```

key

- Allows the creation of multiple versions of a dynamic region, each using different runtime constants
- Separate code is dynamically generated for each distinct combination of values of the runtime constants

```
      }
      return CacheMiss;
   } /* end of dynamic region */
}
```

May 4, 2015 Loop Transformations 41

The Need for Annotations

Automatic dynamic compilation is difficult

- Which variables are runtime constant over which pieces of code?
- Complicated by aliases, side effects, pointers that can modify memory
- Which loops are profitable to unroll?
- Estimating **profitability** is the difficult part

Annotation errors

- Lead to incorrect dynamic compilation
  - *e.g.*, Incorrect code if a value is not really a runtime constant

May 4, 2015 Loop Transformations 42

The Static Compiler

Operates on low-level IR

- CFG + three address code in SSA form

Tasks

- Identifies runtime constants inside of **dynamic regions**
- Splits each **dynamic region** subgraph into **set-up** and **template** code subgraphs
- Optimizes the control flow for each procedure
- Generates machine code, including **templates**
  - In most cases, table space for runtime constants can be statically allocated
  - What do we do about unrolled loops?
- Generates stitcher **directives**

Detecting Runtime Constants

Simple data-flow analysis

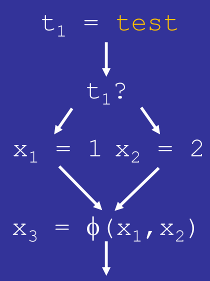
- Propagates initial runtime constants through the dynamic region using the following transfer functions

– x = y	x is a constant iff y is a constant
– x = y op z	x is a const iff y and z are consts and op is an idempotent, side-effect free, non-trapping op
– x = f(y_1, … y_n)	x is a const iff the y_i are consts and f is an idempotent, side-effect free, non-trapping function
– x = *p	x is a constant iff p is constant
– x = dynamic *p	x is not constant

Detecting Runtime Constants (cont)

Merging control flow

– If a variable has the same runtime constant reaching definition along all predecessors, it's considered a constant after the merge



– If test is a runtime constant, then we'll always take one branch or the other

– If test is constant, ϕ is idempotent so the result is constant

– If test is not constant, ϕ is not idempotent, so the result is not constant

Optimizations

Integrated optimizations

– For best quality code, optimizations should be performed across dynamic region boundaries, *e.g.*, global CSE, global register allocation

– Optimizations can be performed both before and after the dynamic region has been split into setup and template codes

Restrictions on optimizing split code

– Instructions with holes cannot be moved outside of their dynamic region

– Holes cannot be treated as legal values outside of the dynamic region. (*e.g.*, Copy propagation cannot propagate values of holes outside of dynamic regions)

– Holes are typically viewed as constants throughout the dynamic region, but induction variables become constant for only a given iteration of an unrolled loop

The Stitcher

Performs directive-driven tasks
- Patches holes in templates
- Unrolls loops
- Patches pc-relative instructions (such as relative branches)

Performs simple peephole optimizations
- Strength reduction of multiplies, unsigned division, modulus

The End Result

Final dynamically generated code from our example
- Assuming the following configuration:
  - 512 lines, 32 byte blocks, 4-way set associative
  - `cache` is an address loaded from the runtime constants table

```
cacheResult cacheLookup (void *addr, Cache *cache) {
      int gat = addr >> 14;
      int line = (add >> 5) & 511;
      setStructure **setArray = cache->lines[line]->sets;
      if (setArray[0]->tag == tag) goto L1:
      if (setArray[1]->tag == tag) goto L1:
      if (setArray[2]->tag == tag) goto L1:
      if (setArray[3]->tag == tag) goto L1:
      return CacheMiss;
L1:   return CacheHit;
```

The Original Code without Annotations

```
cacheResult cacheLookup (void *addr, Cache *cache) {
      int blockSize = cache->blockSize;
      int numLines = cache->numLines;
      int tag = addr / (blockSize * numLines);
      int line = (add / blockSize) % numLines;
      setStructure **setArray = cache->lines[line]->sets;
      int assoc = cache->associativity;
      int set;

      for (set=0; set<assoc; set++) {
          if (setArray[set]->tag == tag)
              return CacheHit;
      }
      return CacheMiss;
}
```


Performance Results

Two measures of performance

- **Asymptotic improvement**: speedup if overhead were 0
- **Break even point**: the fewest number of iterations at which the dynamic compilation system is profitable

Benchmark	Asymptotic speedup of dynamic regions	Breakeven point
calculator	1.7	916 interpretations
matrix multiply	1.6	31,392 scalar ×'s
sparse mat multiply	1.8	2645 matrix ×'s
event dispatcher	1.4	722 event dispatches
quicksort	1.2	3050 records

Evaluation

Today's discussion

- Simple caching scheme
  - Setup once, reuse thereafter
- More sophisticated schemes are possible
  - Can cache multiple versions of code
  - Can provide eager, or speculative, specialization
  - Can allow different dynamic regions for different variables

Subsequent progress on DyC

- More sophisticated language and compiler [Grant, *et al* 1999]
  - More complexity is needed
  - Extremely difficult to annotate the applications
- Automated insertion of annotations [Mock, *et al* 2000]
  - Use profiling to obtain value and frequency information

May 4, 2015 Loop Transformations 51