MLIR for Fortran

Vivek Sarkar * (Georgia Tech)
Nelson Amaral (U.Alberta)
Kit Barton (IBM)
Wang Chen (IBM)

* Disclaimer: opinions in this talk are those of the presenter, and do not reflect the official opinions of IBM!

Overview

- New research project planned for 2020 across Georgia Tech, U.Alberta, IBM
- Motivated by past work at IBM on ASTI optimizer for Fortran, and recent MLIR work at IBM with U.Alberta
- Goal is to leverage past experiences with ASTI's High-level Intermediate Representation (HIR) to evaluate the design space for an MLIR dialect for optimization of Fortran codes
  - Special focus on array statements, loops, array accesses
  - Such an MLIR dialect could be useful for optimization of HPC codes in other languages as well
- Complementary to ECP Flang project
- All feedback and suggestions are most welcome!

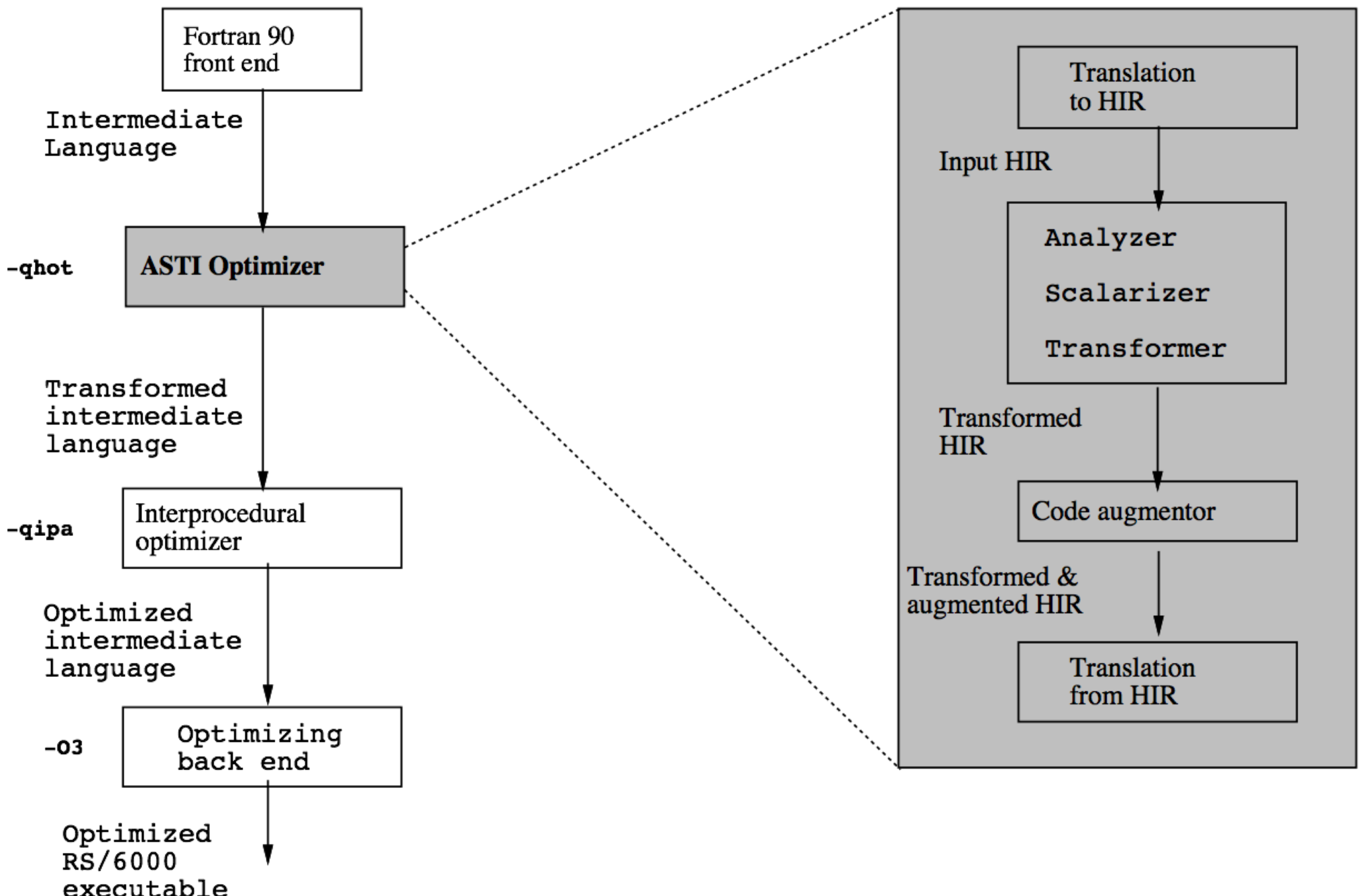
Background: ASTI Optimizer

(Analyzer, Scalarizer, Transformer, Interprocedural optimizer)

- Automatic selection of high-order transformations in the IBM XL Fortran compilers
- Quantitative approach to program optimization using cost models
- High-order transformations selected for *uniprocessor* target include: loop distribution, fusion, interchange, reversal, skewing, tiling, unrolling, and scalar replacement of array references
- Design and initial product implementation completed during 1991–1993

Reference: "Automatic Selection of High Order Transformations in the IBM XL Fortran Compilers", V. Sarkar, IBM Journal of Res. & Dev., Vol. 41, No. 3, May 1997.

Structure of XL Fortran Product Compiler (Version 4)



Scalarization example

```
(a) Example loop containing an array assignment statem
------------------------------------------------------
do k = 1, n
   A(1:n,1:n) = A(1:n,1:n) + spread(B(1:n,k),2,n) *
                             spread(C(k,1:n),1,n)
end do

(b) After unoptimized scalarization:
------------------------------------
do k = 1, n
   do i1 = 1, n  ! parallelizable loop
      T1(i1) = B(i1,k)
   end do

   do i2 = 1, n  ! parallelizable loop
      do i1 = 1, n  ! parallelizable loop
         T2(i1,i2) = T1(i1)
      end do
   end do

   do i1 = 1, n  ! parallelizable loop
      T3(i1) = C(k,i1)
   end do

   do i2 = 1, n  ! parallelizable loop
      do i1 = 1, n  ! parallelizable loop
         T4(i1,i2) = T3(i2)
      end do
   end do

   do i2 = 1, n  ! parallelizable loop
      do i1 = 1, n   ! parallelizable loop
         T5(i1,i2) = A(i1,i2) + T2(i1,i2) * T4(i1,i2)
      end do
   end do

   do i2 = 1, n  ! parallelizable loop
      do i1 = 1, n   ! parallelizable loop
         A(i1,i2) = T5(i1,i2)
      end do
   end do
end do
```

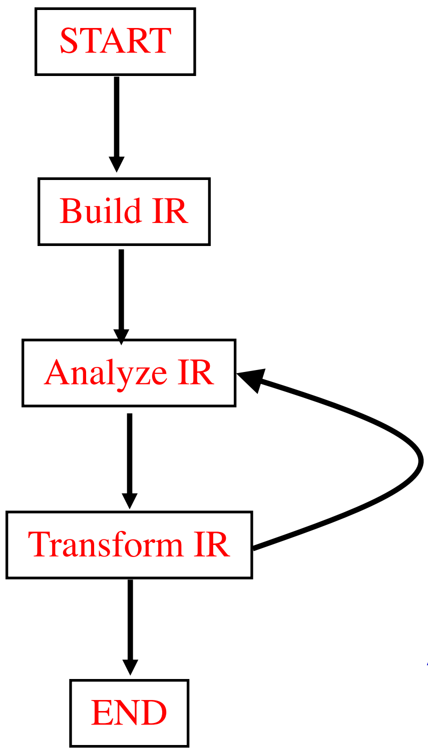
```
(c) After optimized scalarization:
----------------------------------
do k = 1, n
   do i2 = 1, n  ! parallelizable loop
      do i1 = 1, n   ! parallelizable loop
         A(i1,i2) = A(i1,i2) + B(i1,k) * C(k,i2)
      end do
   end do
end do

(d) After collective transformation of loop nest (c):
-----------------------------------------------------
do bb$_i2=1,n,b$_i2   ! parallelizable loop
   do bb$_i1=1,n,b$_i1   ! parallelizable loop
      do bb$_k =1,n,b$_k
         do i2=max(1,bb$_i2),min(n,bb$_i2+b$_i2-1)
            do i1=max(1,bb$_i1),min(n,bb$_i1+b$_i1-1)
               do k=max(1,bb$_k),min(n,bb$_k+b$_k-1),1
                  A(i1,i2) = A(i1,i2) + B(i1,k) * C(k,i2)
               end do
            end do
         end do
      end do
   end do
end do
```

Structure of ASTI Transformer

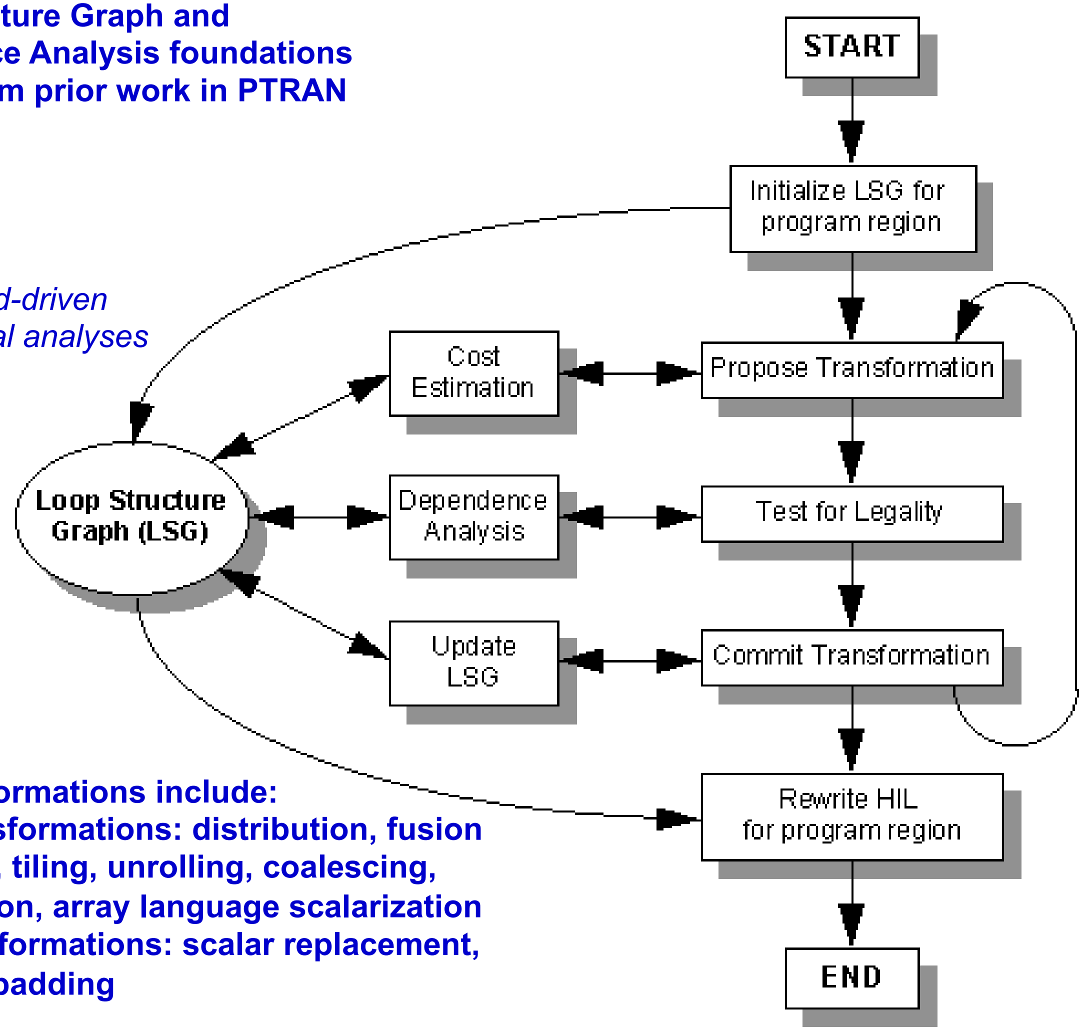
[LCPC 1991, PLDI 1992, CASCON 1994, ICPP 1995, IBM JRD 1997, ICPP 1997, SPAA 1997, LCR 1998, LCPC 1998, ISPASS 2000, ICS 2000, IJPP 2001]

Traditional Optimizer Structure



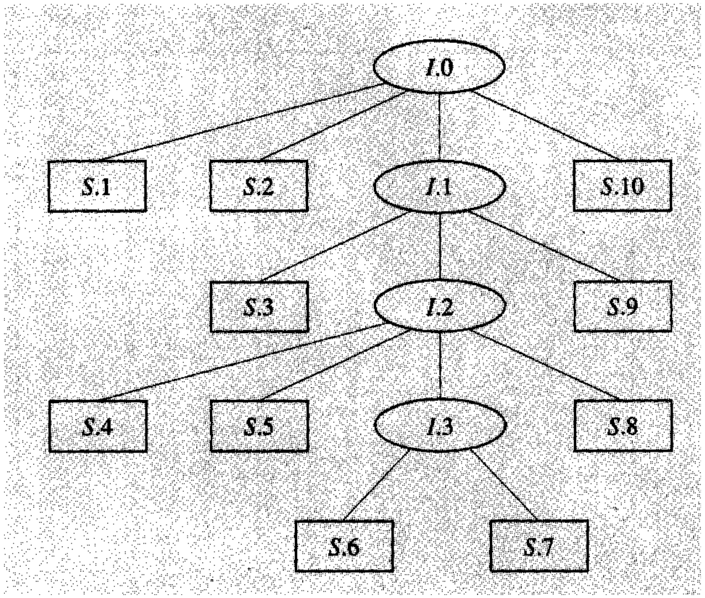
Loop Structure Graph and Dependence Analysis foundations derived from prior work in PTRAN project

Demand-driven incremental analyses

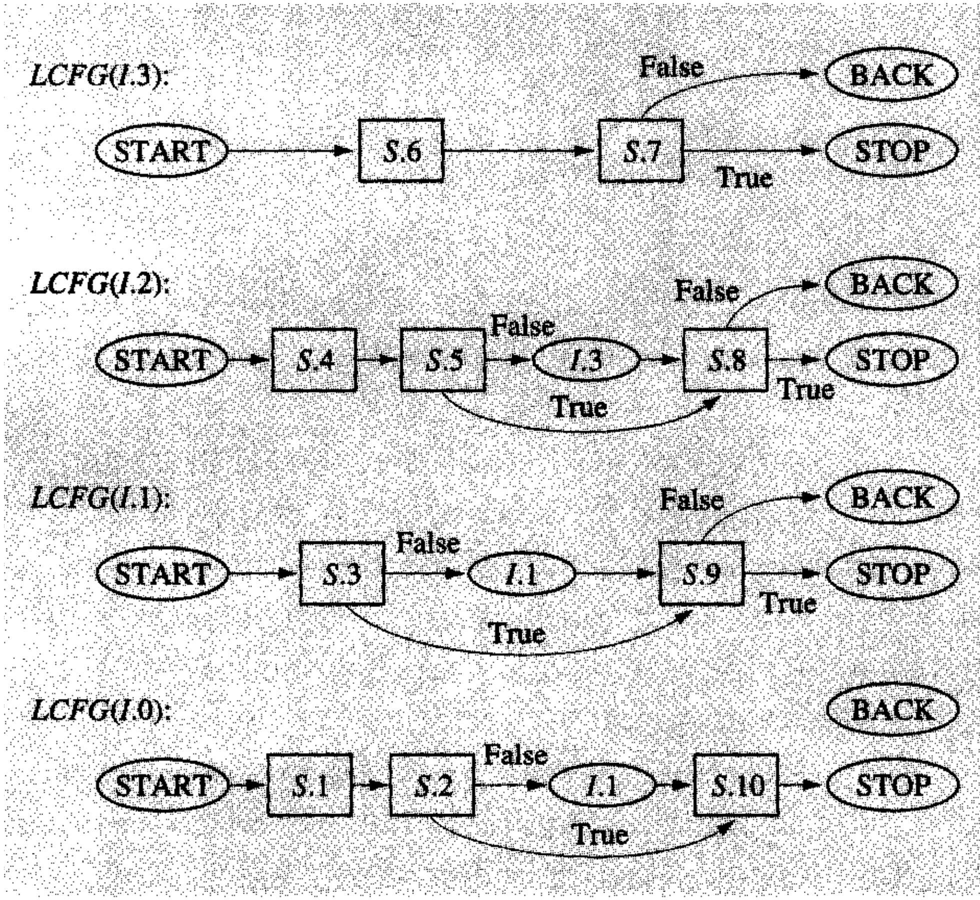


ASTI transformations include:
- **Loop transformations: distribution, fusion unimodular, tiling, unrolling, coalescing, parallelization, array language scalarization**
- **Data transformations: scalar replacement, alignment, padding**

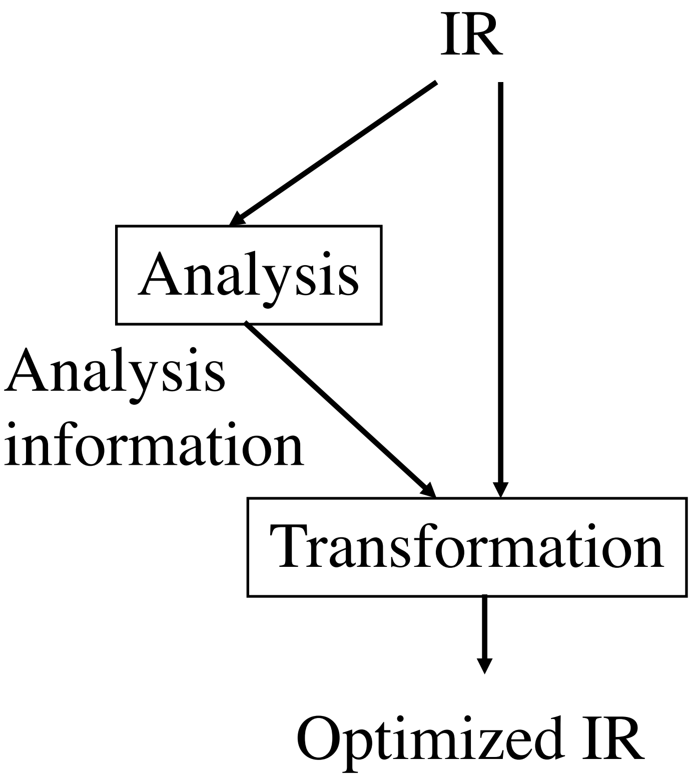
Loop Structure Tree



Loop-Level Control Flow Graphs



Structure of a Single Optimization Pass



Examples:

Analysis	**Transformation**
Value numbering	Common subexpression elimination
Liveness analysis	Dead store elimination
Dependence analysis	Instruction scheduling

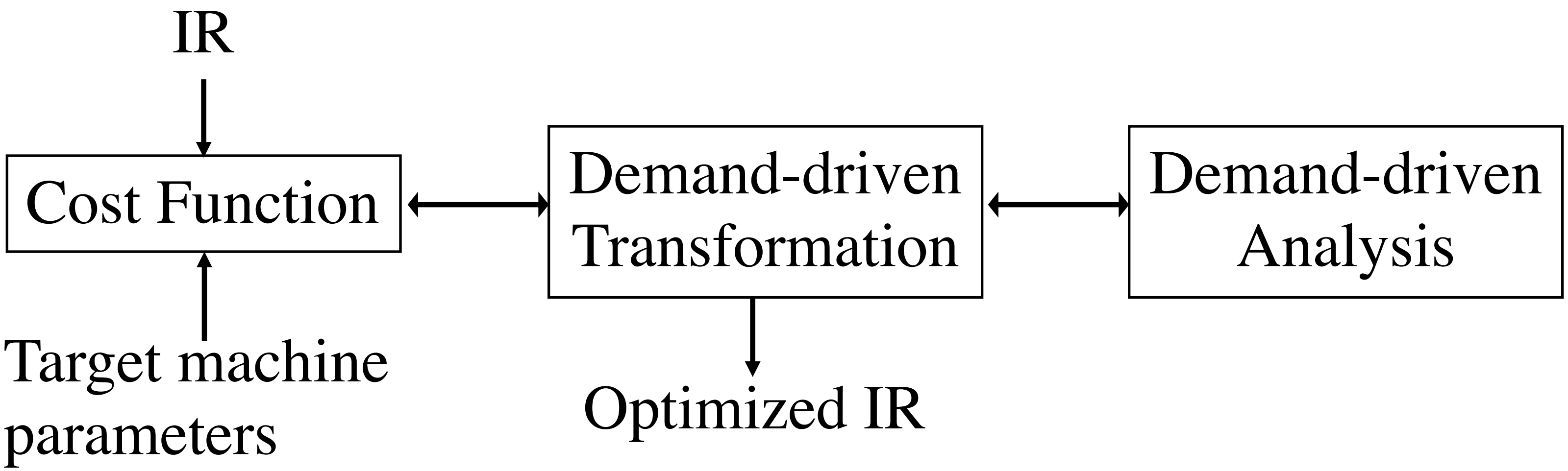
Optimization = Analysis + Transformation

Issues

- **Heuristics:** sequence of optimization passes for a given optimization level is usually hardwired
- **Phase ordering:** a later optimization can create new opportunities for a previous optimization
- **Compile time/space overheads**: all optimization passes are performed on all instructions in all procedures
- **Unpredictability:** hard to predict how much performance improvement will be delivered by compiler optimizations
- **Pessimization:** optimizing transformations can sometimes degrade performance

Rethinking compilers …a quantitative approach

Optimization = Cost function + Analysis + Transformation



Examples:

Cost Function	**Analysis**	**Transformation**
Store freqs	Live variable analysis	Dead store elimination
Basic block freqs	Bounds analysis	Bounds check elimination
Cache misses	Dependence vectors	Loop interchange and tiling

Transformations performed in ASTI for Locality Optimizations

1. Initialization
2. Loop distribution
3. Identification of perfect loop nests
4. Reduction recognition
5. Locality optimization
6. Loop fusion
7. Loop–invariant scalar replacement
8. Loop unrolling and interleaving
9. Local scalar replacement
10. Transcription — generate transformed HIR

Other transformations performed for Vector/SMP/HPF parallelization

Challenges

- Cost function should be efficient to compute but sufficiently accurate
  - Use *lower and upper bounds* as approximations
- Well-tuned code should not incur large compilation overhead
  - Perform analysis and transformation *incrementally* and *on demand*, only when cost function indicates potential for performance improvement
  - Use algorithms with low-polynomial-time complexity
- Phase ordering should be driven by cost functions
  - Use classical optimization theory heuristics in driver for optimizing compiler e.g., sort potential transformations in decreasing order of benefit

Summary

- Time to rethink optimizing compilers
  - Reduce compilation overhead
  - Increase optimization effectiveness
- Quantitative approach provides a promising foundation
- ***Future goal:*** build an optimizer in which all optimization selection and phase ordering decisions are driven by cost functions rather than hardwired heuristics
  - MLIR offers a promising opportunity for such an approach
  - Fortran is an important domain for demonstrating such an approach

BACKUP SLIDES START HERE

Structure of Optimizing Compilers