Compiler Technology for Scientific Computing

David A. Padua

University of Illinois at Urbana-Champaign

Compiler Technology and Performance

- Although clock rate gains have been impressive, architectural features have also contributed significantly to performance improvement.
- At the instruction level, compilers play a crucial role in exploiting the power of the target machine.

Compiler Technology and Performance (Cont.)

- And, with SMT and SMP machines growing in popularity, automatic transformation and analysis at the thread level will become increasingly important.

Compiler Research

- Advances in compiler technology can have a profound impact
  - By improving performance.
  - By facilitating programming without hindering performance.
- For example, average performance improvement of advanced compiler techniques at the instruction level for the IA-64 is ~3. *This is beyond what can be achieved by traditional optimizations*.

Compiler Research and Scientific Computing

- Scientific Computing has had a very important influence on compiler technology.
  - Fortran I (1957)
  - Many classical optimizations
    - strength reduction
    - Common subexpression elimination
    - …
  - Dependence analysis

Compiler Research and Scientific Computing (Cont.)

- Relative commercial importance of scientific computing is declining.
  - End of Cold War
  - Increasing volume of “non-numerical” applications

Compiler Techniques

- Every optimizing compiler does:
  - Program analysis
  - Transformations
- The result of the analysis determines which transformations are possible, but the number of possible valid transformations is unlimited.
- Implicit or explicit heuristics are used to determine which transformations to apply.

Compiler Techniques (Cont.)

- Program analysis is well understood for some classes of computations such as simple dense computations.
- However, compilers usually do a poor job in some cases, especially in the presence of irregular memory accesses.
- Program analysis and transformation is built on a solid mathematical foundation, but building the engine that drives the transformations is an obscure craft.

Outline of the Talk

- Compiler Techniques for Very High Level Languages
- Advanced Program Analysis Techniques
- Static Performance Prediction and Optimization Control

Compiler Techniques for Very High Level Languages

1. Compiling Interactive Array Languages (MATLAB)

Joint work with

George Almasi (Illinois), Luiz De Rose (IBM Research), Vijay Menon (Cornell), Keshav Pingali (Cornell)

Motivation

- Many computational science applications involve matrix manipulations.
- It is, therefore, appropriate to use array languages to program them.
- Furthermore, interactive array languages like APL and MATLAB are very popular today.

Motivation *(continued)*

- Their environment includes easy to use facilities for input of data and display of results.
- Interactivity facilitates debugging and development.
  - Increased software productivity.
- But interactive languages are usually interpreted.
  - Performance suffers.

Long Term Project Goals

- Generation of high-performance code for serial and parallel architectures
- Efficient extraction of information from the high-level semantics of MATLAB
- Use of the semantic information in order to improve compilation

Problem Overview

- MATLAB does not require declarations
  - Static and dynamic type inference
- Variables can change characteristics during execution time

Type Inference

- Type inference needs to determine the following variable properties:
  - Intrinsic type: logical, integer, real, or complex
  - Rank: scalar, vector, or matrix
  - Shape: size of each dimension
- For more advanced transformations (not implemented) it also will be useful to determine:
  - Structure: lower triangular, diagonal, etc

Test Programs

Test programs		*Problem size*	*Source*
Successive Overrelaxation	***(SOR)***	420x420	a
Preconditioned CG	***(CG)***	420x420	a
Generation of a 3D-Surface	***(3D)***	51x31x21	d
Quasi-Minimal Residual	***(QMR)***	420x420	a
Adaptive Quadrature	***(AQ)***	1Dim (7)	b
Incomplete Cholesky Factorization	***(IC)***	400x400	d
Galerkin (Poisson equation)	***(Ga)***	40x40	c
Crank-Nicholson (heat equation)	***(CN)***	321x321	b
Two body problem 4th order Runge-Kutta	***(RK)***	3200 steps	c
Two body problem Euler-Cromer method	***(EC)***	6240 steps	c
Dirichlet (Laplace's equation)	***(Di)***	41x41	b
Finite Difference (wave equation)	***(FD)***	451x451	b

Source:

a. "Templates for the Solution of Linear Systems Building Blocks for Iterative Methods", Barrett et. Al.

b. "Numerical Methods for Mathematics, Science and Engineering, John H. Mathews

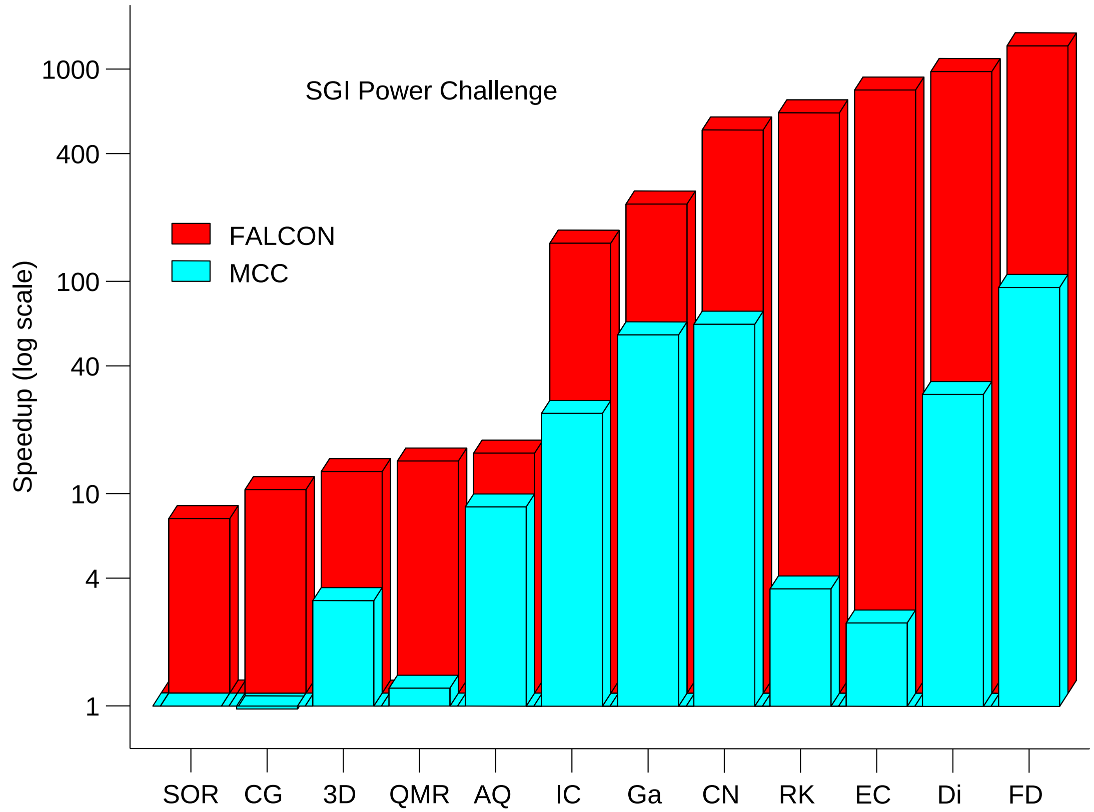
c. "Numerical Methods for Physics", A. Garcia

d. Colleagues

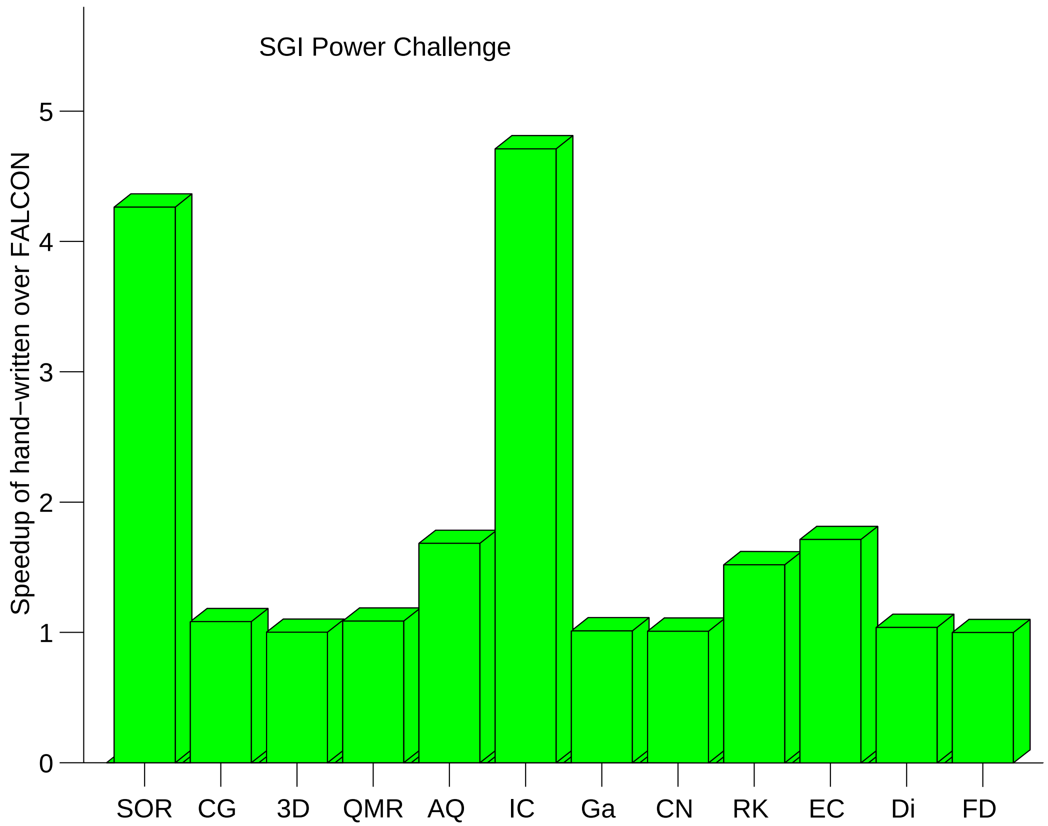
Times on the SGI Power Challenge

Prog.	MATLAB	MCC	F 90	H. W.
SOR	18.12	18.14	2.733	0.641
CG	5.34	5.51	0.588	0.543
3D	34.95	11.14	3.163	3.158
QMR	7.58	6.24	0.611	0.562
AQ	19.95	2.30	1.477	0.877
IC	32.28	1.35	0.245	0.052
Ga	31.44	0.56	0.156	0.154
CN	44.10	0.70	0.098	0.097
RK	20.60	5.77	0.038	0.025
EC	8.34	3.38	0.012	0;007
Di	44.17	1.50	0.052	0.050
FD	34.80	0.37	0.031	0.031

Speedups on the SGI (1 Processor)



Hand-Written Code Comparison



MAJIC Overview

- MAJIC: (MAtlab Just-In-Time Compiler) interactive and fast
  - combination interpreter/JIT compiler
  - builds on top of FALCON techniques

Compiler Techniques in MAJIC

Analysis

- Compile only code that takes time to execute (loops)
- type analysis and value/limit propagation
- recompile only when source has changed

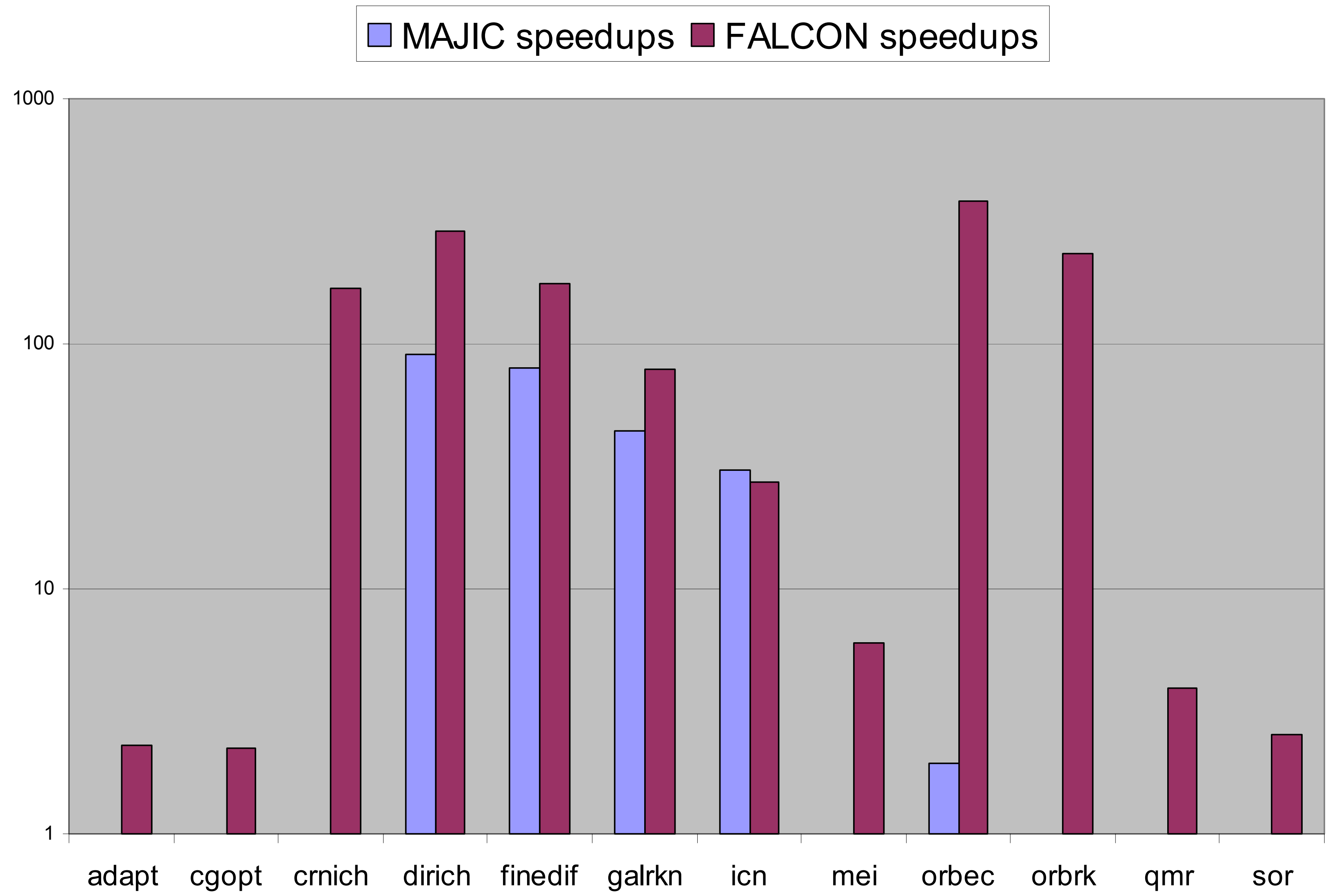
Code Generation

- naïve (per AST node) JIT code generation
- uses built-in MATLAB functions where possible
- average compile time: 20ms per line of MATLAB source

Future Compiler Techniques

- JIT SMP parallelism: lightweight dependence analysis, execution on multiple Solaris LWPs
- speculative lookahead compilation: hiding compilation time from user
- exact shape/value propagation: allows precise unrolling for “small vector” operations

Current Results (Feb 2000)



Compiler Techniques for Very High Level Languages

2. Compiling Tensor Product Language

Joint work with

Jeremy Johnson (Drexel), Robert Johnson (MathStar), Jose Moura (CMU), Viktor Prasanna (SC), Manuela Veloso (CMU)

Cooley-Tukey Theorem and the FFT

In 1964, Cooley and Tukey presented a divide and conquer algorithm for computing $f=F_n x$. Their algorithm is based on

Theorem. Let $n=rs$, $0 \le k_1, l_2 < r, 0 \le k_2, l_1 < s$, then

$$y(l_1 + l_2 s) = \sum_{k_1} w^{k_1 l_2 s} (w^{k_1 l_1} (\sum_{k_2} x(k_1 + k_2 r) w^{k_2 l_1 r}))$$

Repeated application to $n=2^t$, for example, leads to an $O(n \log n)$ algorithm, called the Fast Fourier Transform (FFT), for computing $f=F_n x$.

Tensor Product of Matrices

Let A be an $m\times m$ matrix and B an $n\times n$ matrix. Then $A\otimes B$ is the $mn\times mn$ matrix defined by the block matrix product

$$A \otimes B = \left(a_{i,j}B\right)_{1\le i,j\le m} = \begin{pmatrix} a_{1,1}B & \cdots & a_{1,m}B \\ \vdots & \ddots & \vdots \\ a_{m,1}B & \cdots & a_{m,m}B \end{pmatrix}$$

For example, if

$$A = \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \quad \text{and} \quad B = \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix}$$

then

$$A \otimes B = \begin{bmatrix} a_{11}b_{11} & a_{11}b_{12} & a_{12}b_{11} & a_{12}b_{12} \\ a_{11}b_{21} & a_{11}b_{22} & a_{12}b_{21} & a_{12}b_{22} \\ a_{21}b_{11} & a_{21}b_{12} & a_{22}b_{11} & a_{22}b_{12} \\ a_{21}b_{21} & a_{21}b_{22} & a_{22}b_{21} & a_{22}b_{22} \end{bmatrix}$$

Tensor Product Formulation of Cooley-Tuckey

Theorem $F_{rs} = (F_r \otimes I_s)T_s^{rs}(I_r \otimes F_s)L_r^{rs}$

T_s^{rs} is a diagonal matrix

L_r^{rs} is a stride permutation

Example

$$F_4 = (F_2 \otimes I_2)T_4^4(I_2 \otimes F_2)L_2^4$$

$$= \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & -1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 & 0 & 0 \\ 1 & -1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & -1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Properties of the Tensor Product

Associativity

$$(A \otimes B) \otimes C = A \otimes (B \otimes C)$$

Multiplicative Property

$$AC \otimes BD = (A \otimes B)(C \otimes D)$$

$$A \otimes B = (A \otimes I_n)(I_m \otimes B)$$

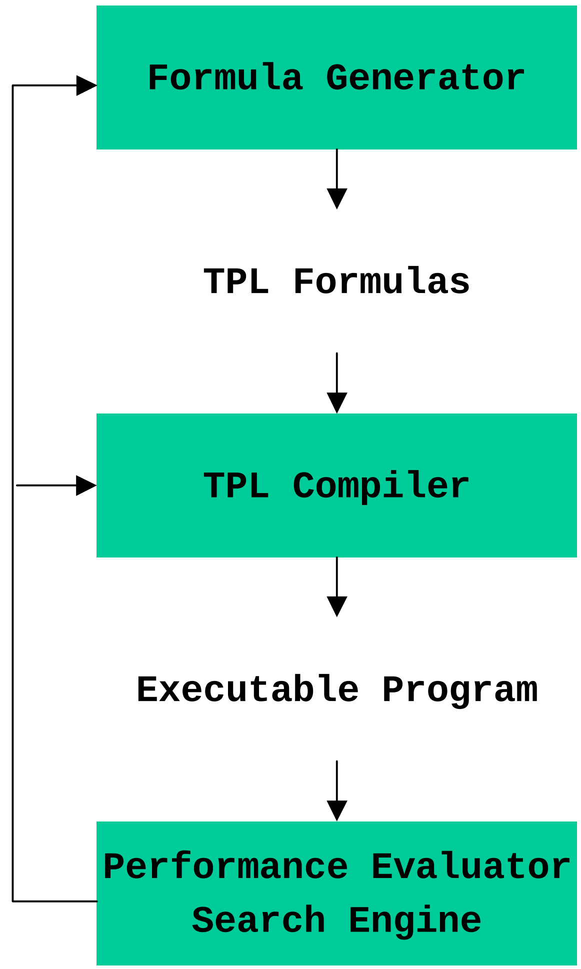
Commutation Theorem

$$A \otimes B = L_m^{mn}(B \otimes A)L_n^{mn}$$

Stride Permutations

$$L_s^{rst} L_t^{rst} = L_{st}^{rst}$$

TPL Compiler



TPL Program

```
#subname F4
  ( compose
        ( tensor (F 2) (I 2) )
        ( T 4 2 )
        ( tensor (I 2) (F 2) )
        ( L 4 2 ) )
```

Unrolled Program

```
subroutine F4(y,x)
implicit complex*16(f)
complex*16 y(4),x(4)

f0   =  x(1) + x(3)
f1   =  x(1) - x(3)
f2   =  x(2) + x(4)
f3   =  x(2) - x(4)
f4   =  (0.0d0,-1.0d0) * f3
y(1) =  f0 + f2
y(3) =  f0 - f2
y(2) =  f1 + f4
y(4) =  f1 - f4
end
```

Loop Version

```
subroutine F4L(y,x)
implicit complex*16(f)
complex*16 y(4),x(4),t1(4)

do i0 = 0, 1
     t1(2*i0+1)       =  x(i0+1) + x(i0+3)
     t1(2*i0+2)       =  x(i0+1) - x(i0+3)
end do

t1(4)=  (0.0d0,-1.0d0) * t1(4)

do i0 = 0, 1
     y(i0+1)  =  t1(i0+1) + t1(i0+3)
     y(i0+3)  =  t1(i0+1) - t1(i0+3)
end do

end
```

Advanced Program Analysis Techniques

1. Analysis of Explicitly Parallel Programs

Joint work with

Samuel Midkiff (IBM Research and Jaejin Lee (Michigan State)

J. Lee, D. Padua, and S. Midkiff. Basic Compiler Algorithms for Parallel Programs. ACM SIGPLAN 1999 Symposium on Principles and Practice of Parallel Programming (PPoPP).

Shared Memory Parallel Programming

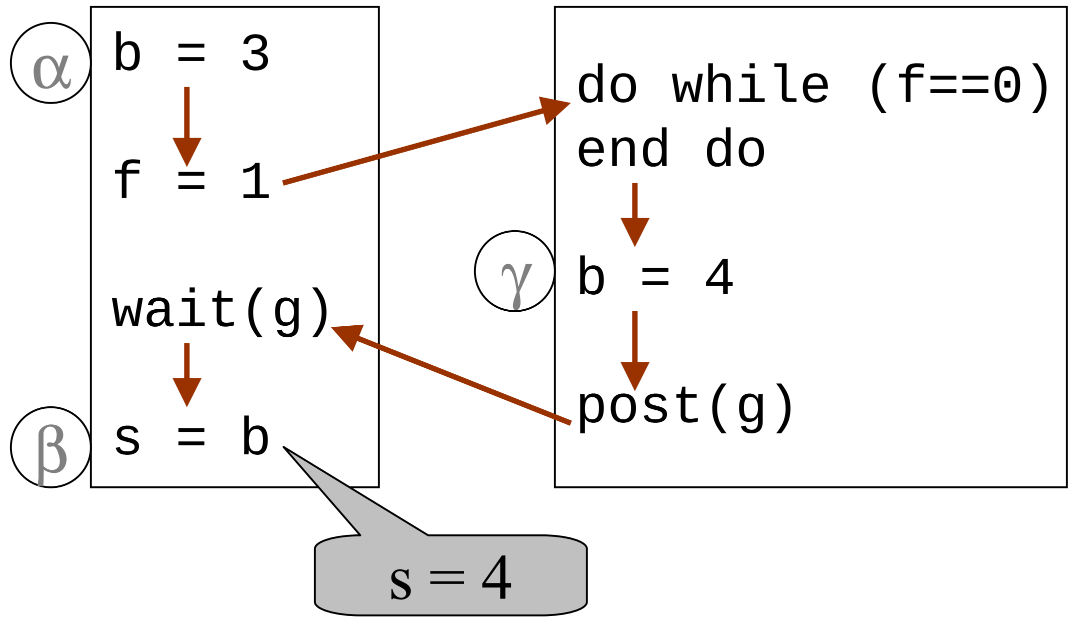
- A global shared address space between threads
- Communication between threads:
  - reads and writes of shared variables

```
flag = 1
```

```
do while (flag==0)
end do
```

Incorrect Constant Propagation

```
b = 0
f = 0
g = 0
s = 0
cobegin
α   b = 3
    f = 1
    wait(g)
β   s = b
||
    do while (f==0)
    end do
γ   b = 4
    post(g)
coend
print s
```



- To avoid this incorrect propagation, we can conservatively assume `wait(g)` modifies variable `b`, but this cannot handle busy-wait synchronization.

Incorrect Loop Invariant Code Motion

```
a = 0
x = 0
y = 0
cobegin
  y = a + 1
  do while (x <> 2)
 α x = a + 1
  end do
||
  a = 1
coend
```

```
x = a + 1
do while (x <> 2)
end do
```

- **`x = a + 1`** is a loop invariant in classical sense, but moving it outside makes the loop infinite.

Incorrect Dead Code Elimination

Thread 1 | Thread 2

α
```
flag = 1
```

```
do While (flag==0)
end do
```

- An instruction *I* is dead if it computes values that are not used in any executable path starting at *I*.
- Eliminating **`flag = 1`** in Thread 1 makes the while loop in Thread 2 infinite.

Incorrect Compilation

```
C$DOACROSS SHARE(a,e,k), LOCAL(I,j)
      do I = 2, n
        do j = 2, n
          do while (e(I-1,j) .ne. 1)
          end do
          a(I,j) = a(I,j-1) + a(I-1,j
          e(I,j) = 1
        end do
      end do
```

W/O optimization

```
$33: R13 ← e(I-1,j)
     if R13 <> 1 goto $33
```

W/ optimization

```
     R13 ← e(I-1,j)
$33:
     if R13 <> 1 goto $33
```

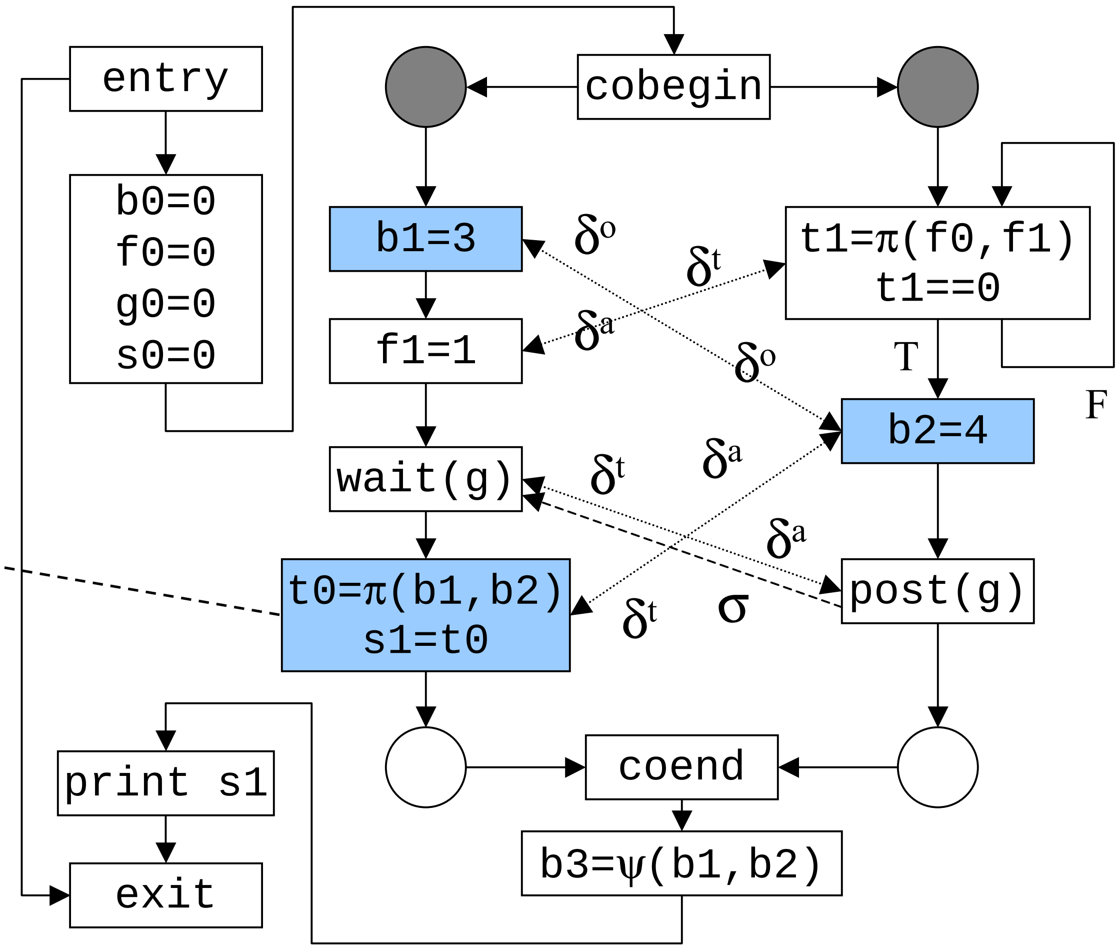
- The compiler makes the busy-wait loop infinite.

Concurrent Static Single Assignment Form (Cont.)

- A π-function of the form $\pi(v_1, \ldots, v_n)$ for a shared variable v is placed where there is a use of the shared variable with δ^t conflict edges. n is the number of reaching definitions to the use of v through the incoming control flow edges and incoming conflict δ^t edges.
- The CSSA form has the following properties:
  – All uses of a variable are reached by exactly one (static) assignment to the variable.
  – For a variable, the definition dominates the uses if they are not arguments of ϕ-, π-, and ψ-functions.

Concurrent Static Single Assignment Form (Cont.)

```
b = 0
f = 0
g = 0
s = 0
cobegin
  b = 3
  f = 1
  wait(g)
  s = b
||
  do while (f==0)
  end do
  b = 4
  post(g)
coend
print s
```



Synchronization Analysis

- We compute a set *Prec*[*n*] of nodes *m*, which is guaranteed to precede *n* during execution.
- Computing the exact execution ordering:
  - NP-hard problem
  - An iterative approximation algorithm
    - Common ancestor algorithm

Common Ancestor Algorithm

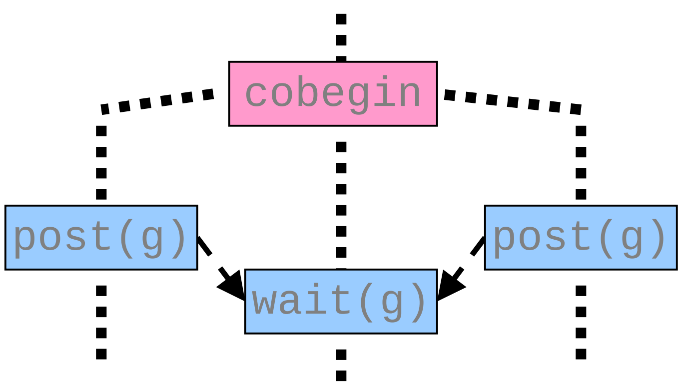
[Emrath,Ghosh, and Padua, IEEE Software, 1992]

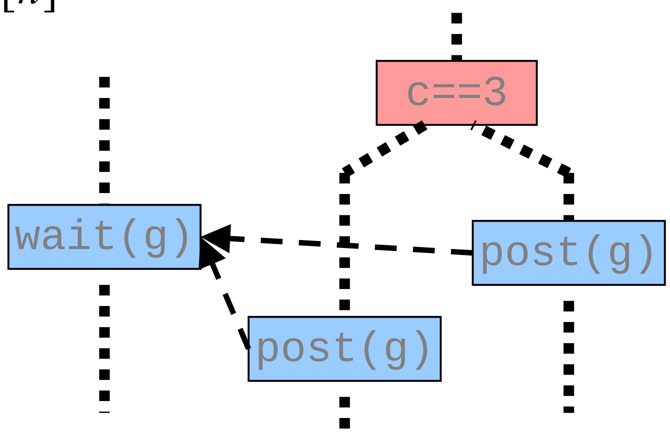
- Originally, a trace based algorithm, but we use it for static analysis.
- Data flow equations:

$$Prec_{cont}[n] = \begin{cases} \bigcup\limits_{(m,n)\in E_{cont}} (Prec[m] \cup \{m\}) & \text{if } n \text{ is a coend node} \\ \bigcap\limits_{(m,n)\in E_{cont}} (Prec[m] \cup \{m\}) & \text{otherwise} \end{cases}$$

$$Prec_{sync}[n] = \bigcap_{(m,n)\in E_{sync}} (Prec[m] \cup \{m\})$$

$$Prec[n] = Prec_{cont}[n] \cup Prec_{sync}[n]$$





Advanced Program Analysis Techniques

2. Analysis of Conventional Programs to Detect Parallelism

Joint work with

W. Blume(HP), J. Hoeflinger (Illinois), R. Eigenmann (Purdue), P. Petersen (KAI), and P. Tu

An Important Area

- Powerful high-level restructuring for parallelism and locality enhancement.
- Language constructs based on early research on detection of parallelism.
- Analysis and transformations important for instruction-level parallelism

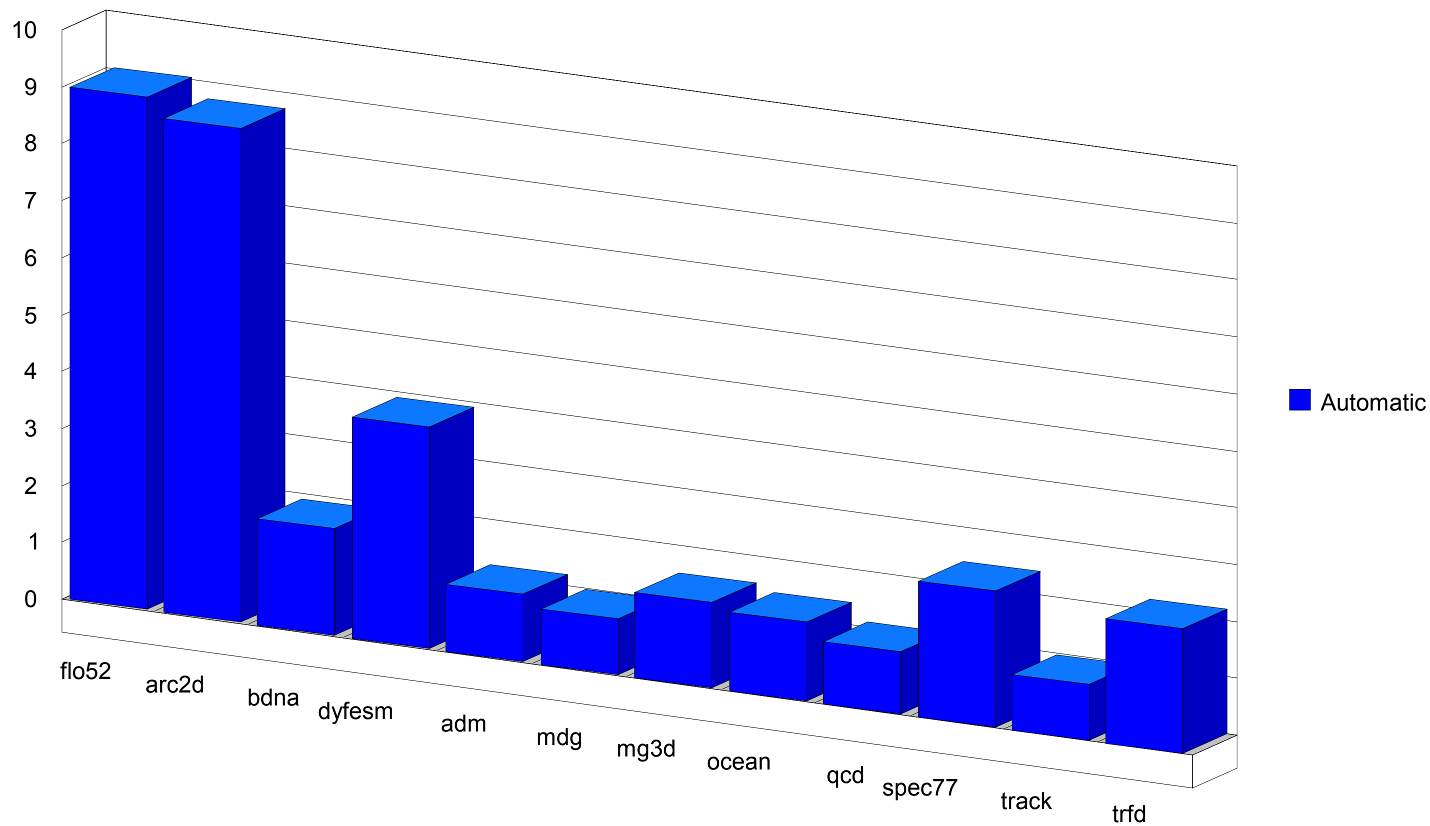
An Experiment on the Alliant FX/80

- In 1989-90, we did experiments on the effectiveness of automatic parallelization for an eight-processor Alliant FX/80 and the Cedar multiprocessor.
- We translated the Perfect Benchmarks using KAP-Cedar, a version of KAP modified at Illinois. The speedups obtained for the Alliant are shown next.

R. Eigenmann, J. Hoeflinger, and D. Padua. *On the Automatic Parallelization of the Perfect Benchmarks*. **IEEE TPDS. 9(1**). 1998.

R. Eigenmann, J. Hoeflinger, Z. Li, and D. Padua. Experience in the Automatic Parallelization of Four Perfect Benchmark Programs. Lecture Notes in Computer Science 589. Springer-Verlag. 1992.

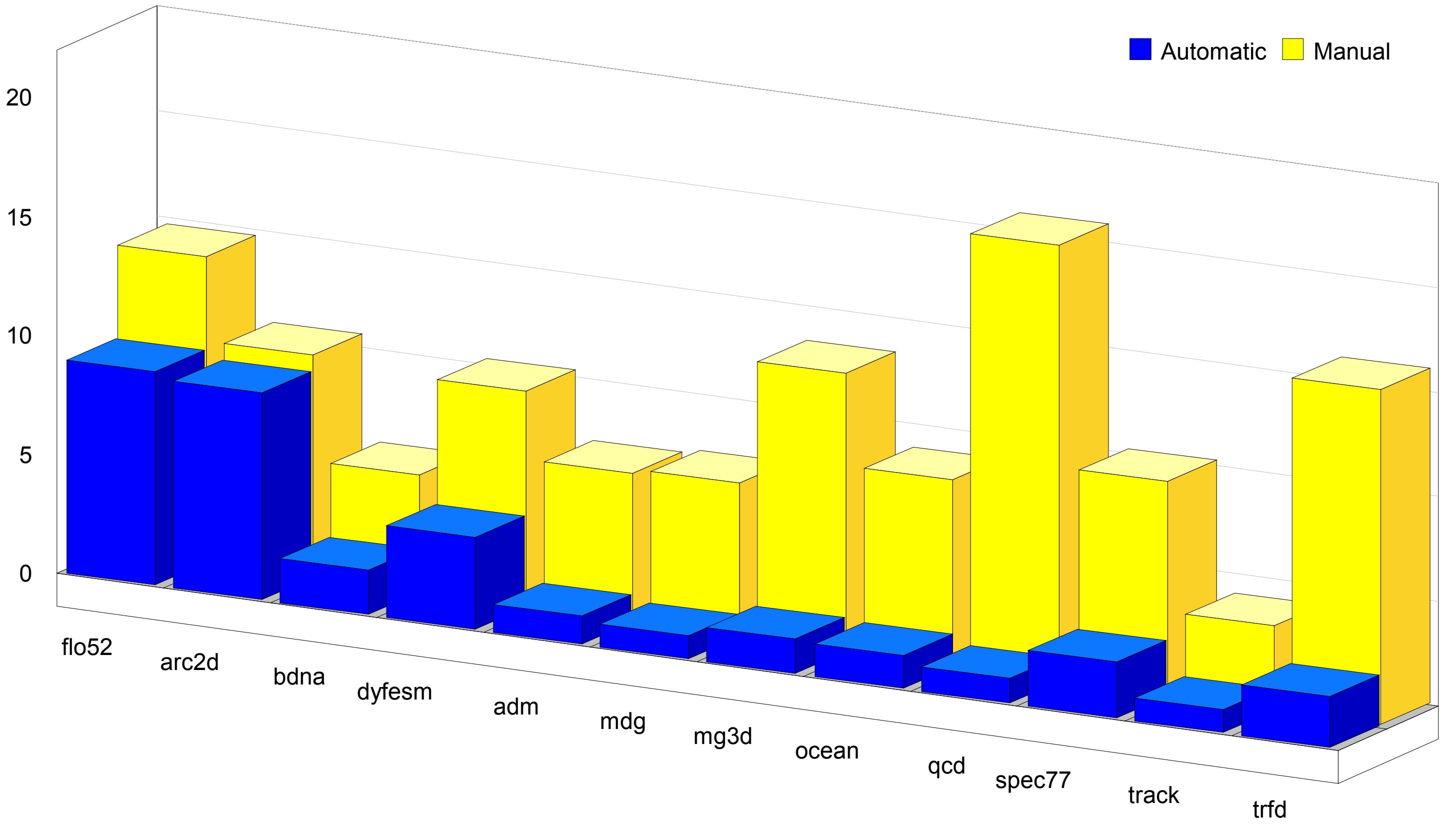
Automatically-obtained speedups on the Alliant FX/80



An experiment on the Alliant FX/80 (cont.)

- As can be seen, the speedups obtained were dismal.
- However, we found that there is much parallelism in the Perfect Benchmarks and, more importantly, that it can be detected automatically.
- In fact, after applying by hand a few automatable transformations, we obtained the following results.

Speedups on the Alliant FX/80



New compiler techniques

- Three classes of automatable transformations were found to be of importance in the previous study:
- Recognition and translation of idioms
- Dependence analysis
- Array privatization
- Transformations similar but weaker than those we applied by hand were applied by KAP.

Dependence Analysis

- Given the program:

```
do i=1,n
        do j=1,m
        S:   A(f(i,j),g(i,j))=...
        T:   ...              =A(p(i,j),q(i,j))+ ...
        end do
end do
```

Dependence Analysis (cont.)

- To determine if there is a flow dependence from S to T, we need to determine whether the system of equations:

$$f(i,j) = p(i',j')$$

$$g(i,j)=q(i',j')$$

has a solution under the constraints

$$n \geq i,\ i' \geq 1,\ m \geq j,\ j' \geq 1, \text{ and } (i',j') \geq (i,j)$$

Dependence Analysis (cont.)

- Loop-carried dependences preclude transformation into parallel form.
- They also preclude some important transformations.
- For example, statement S can be removed from the loop if f(i) is never 2

```
    do i=1,n
S:        A(f(i))=...
T:        Q=A(2)
          ...
   end do
```

Dependence Analysis (cont.)

- Many techniques have been developed to answer this question efficiently.
- Banerjee's test suffices in most situations where the subscript expressions are a linear function of the loop indices.

P.Petersen and D. Padua Static and Dynamic Evaluation of Data Dependence Analysis Techniques. IEEE TPDS. 7(11). Nov. 1996.

Dependence Analysis (cont.)

- However, there are situations where traditional dependence analysis techniques, including Banerjee's test, fail. Two cases we found in our experiments are:
- Nonlinear subscript expressions. These could be present in the original source code or generated by the translator when replacing induction variables.
- Indexed arrays.

Dependence analysis (cont.)

- An example of nonlinear subscript expression from OCEAN:

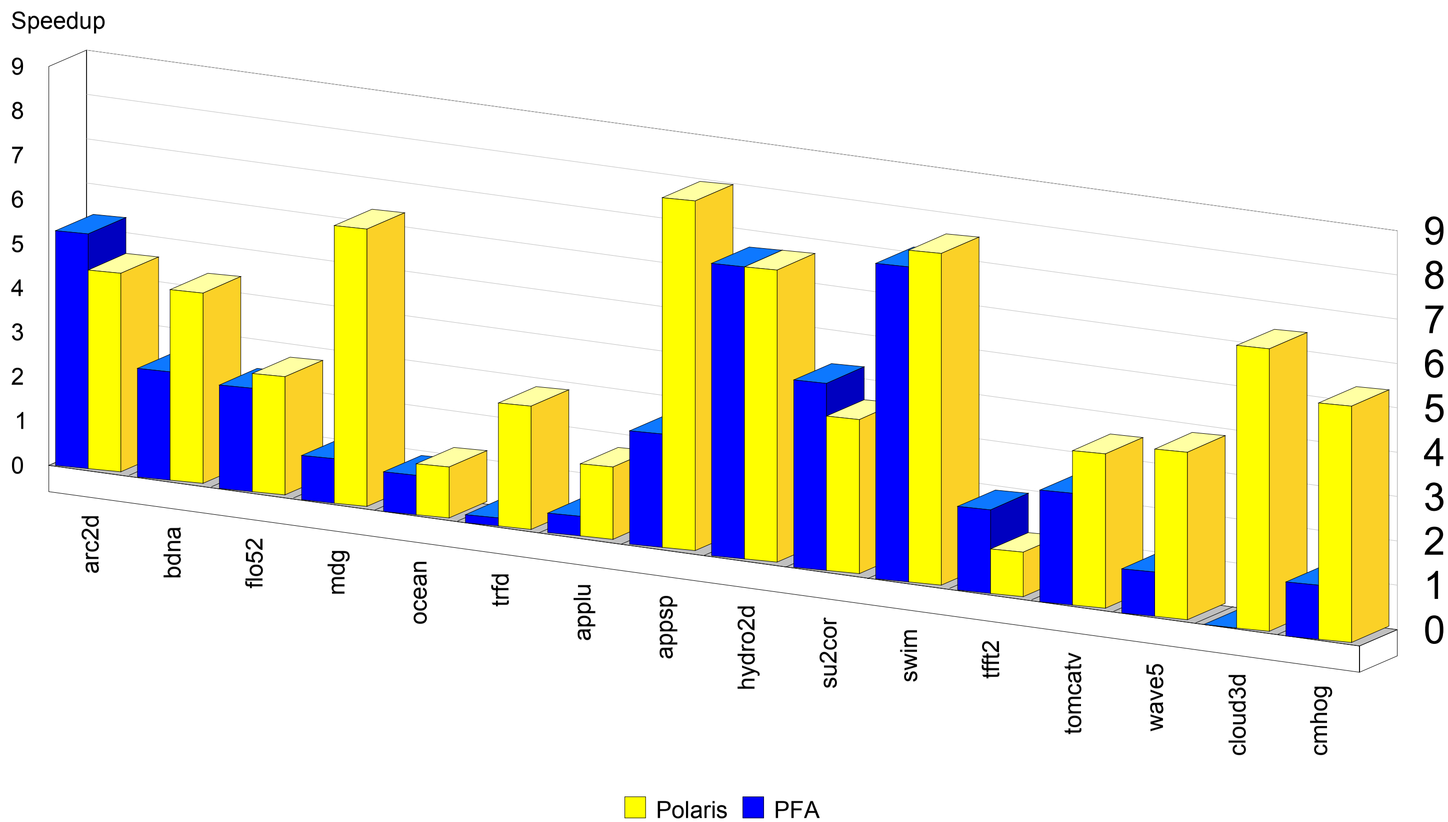
```
do j=0,n-1
      do k=0,x(j)
            do m=0,128
                  ...
                  p=258*n*k+128*j+m+1
                  A(p)=A(p)-A(p+129*n)
                  A(p+129*n)=...
                  ...
```

Dependence analysis (cont.)

- Example of indexed array from TRACK

```
do i=1,n
       ...
       jok= ihits(1,i)
       nused(jok)=nused + 1
       ...
end do
```

Speedups on SGI Challenge (8 processors)



Array privatization

- Each processor cooperating in the execution of a loop has a separate copy of all private variables.
- A variable can be privatized -- that is, replaced by a private variable -- if in every loop iteration the variable is always assigned before it is fetched.

Array privatization (cont.)

- Example of privatizable scalar:

```
do k=...
        s = A(k)+1
        B(k) = s**2
        C(k) = s-1
end do
```

- Example of privatizable array:

```
do k=...
        do j=1,m
                s(j) = A(k,j)+1
        end do
        do j=1,m
                B(k,j) = s(j)**2
                C(k,j) = s(j)-1
        end do
end do
```

Array privatization (cont.)

- Commercial compilers are effective at identifying privatizable scalars.
- However, we found a number of cases where array privatization is necessary for loop parallelization.
- New techniques were developed to deal effectively with array privatization.

Conclusions

- Significant progree is possible
- But few research groups are still focusing on this problem
- An experimental approach is crucial for progress.
  - Need good benchmarks (also desperately needed for research on compilers for explicitly parallel programs)

Advanced Program Analysis Techniques

3. Dependence Analysis

On-going Projects

- Analysis of non-affine subscript expressions.
- Compile-Time analysis of Index Arrays
- Analysis of Java Arrays

Advanced Program Analysis Techniques

3a. Analysis of non-affine subscript expressions

Joint work with

Y. Paek (KAIST) and J. Hoeflinger (Illinois)

Y. Paek, J. Hoeflinger, and D. Padua. Simplification of array access patterns for compiler optimization. SIGPLAN'98 Conference on Programming Language Design and Implementation (PLDI)

Traditional Dependence Analysis

- Traditional data dependence analysis for arrays:
  - form dependence equation
  - solve equation, taking into account constraints

```
DO I=1,N
   A(I) = . . .
   . . .   = A(I+N)
END DO
```

$$I = I' + N$$

Given that $1 <= I <= N$
and $1 <= I' <= N$

What's Wrong with That?

- Tradition is successful, but limited.
  - *coupled subscripts* and *non-affine subscripts* cause problems
  - non-loop parallelization has become important
  - interprocedural parallelization has become important
- Specifically, the system-solving paradigm is too limiting.

Representing an Integer Sequence

- We can precisely represent any regular integer sequence by:
  - its starting value,
  - an expression for the difference between successive elements of the sequence,
  - the total number of elements.

```
do I=1,N
   A(2**I)
end do
```

S.O.S.=2,4,8,16,. . .2^N

Start: 2
$S_{I+1} - S_I = 2^I$
elements: N

LMADs

$$A^{\text{stride } 1, \ldots, \text{stride } d}_{\text{span}_1, \ldots, \text{span}_d} + \text{base offset}$$

Summarizing Loops and CALLs

```
REAL A(N)
DO I=1,N
   A(I) . . .
END DO
```

$$\Rightarrow \quad A_{0}^{0} + I - 1 \quad \underset{\text{expand}}{\Rightarrow} \quad A_{N-1}^{1} + 0$$

```
REAL A(N)
CALL X(A(I))
```

$$\Rightarrow \quad A_{0}^{0} + I - 1 \quad \Rightarrow \quad A_{M-1,\ T*M}^{1,\quad M} + I - 1$$

```
SUBROUTINE X(Z)
REAL Z(*)
 . . .
```

$$\Rightarrow \quad Z_{M-1,\ T*M}^{1,\quad M} + 0$$

Overlap in an LMAD

If a given loop index causes the subscripting offset sequence to produce the same element more than once, then the LMAD is said to have an *overlap* due to that loop index.

This corresponds to a *loop-carried dependence*.

```
do I = 1,N
   do J = 1, M
      A(J) =  . . .
   end do
end do
```

```
Real A(25)
do I = 0,4
   do J = 0, 5
      A(I+3*J) =
   end do
end do
```

Advanced Program Analysis Techniques

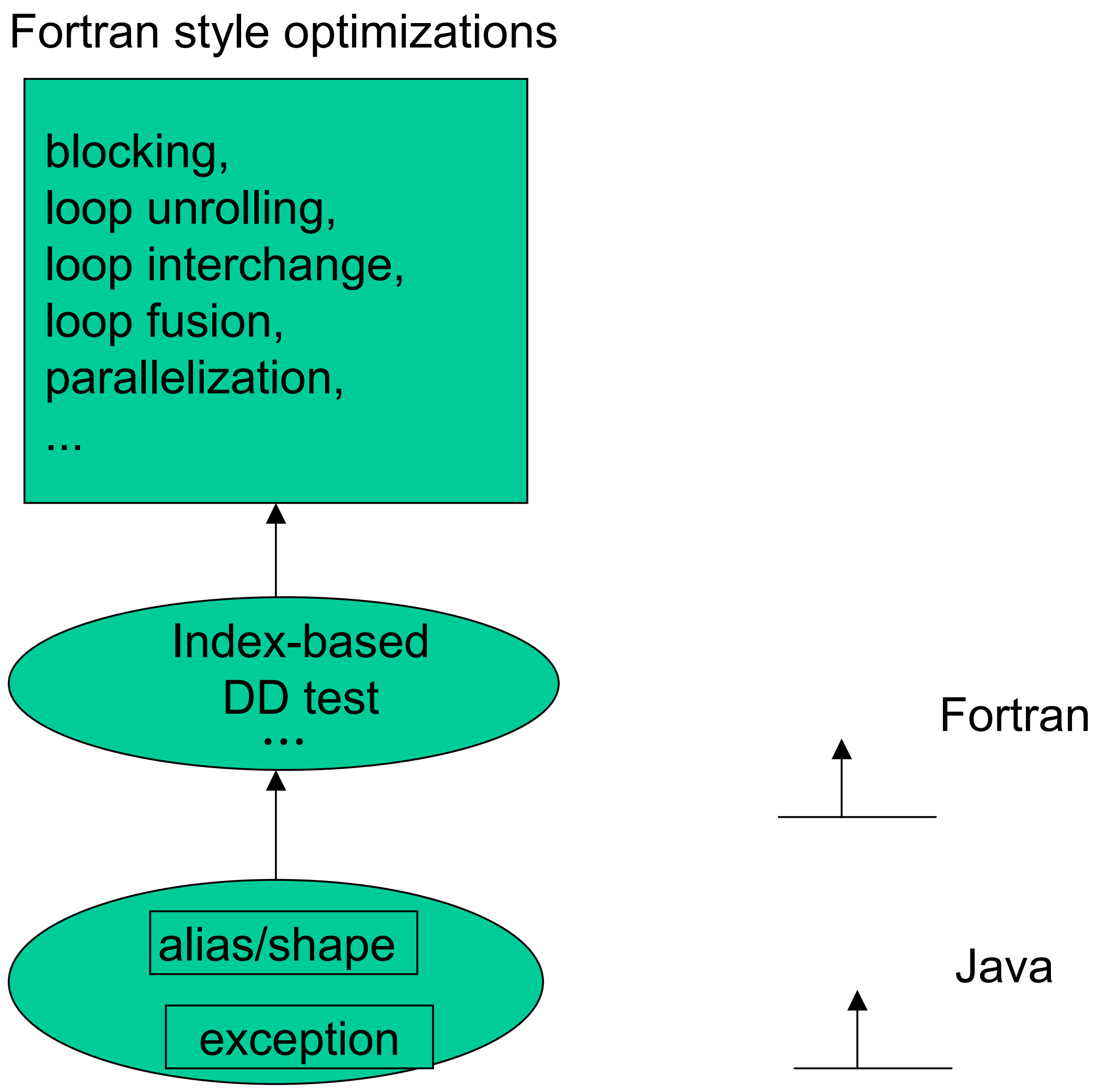
3c. Analysis of Java Arrays

Joint work with

Paul Feautrier (U. Versailles) and Peng Wu (Illinois)

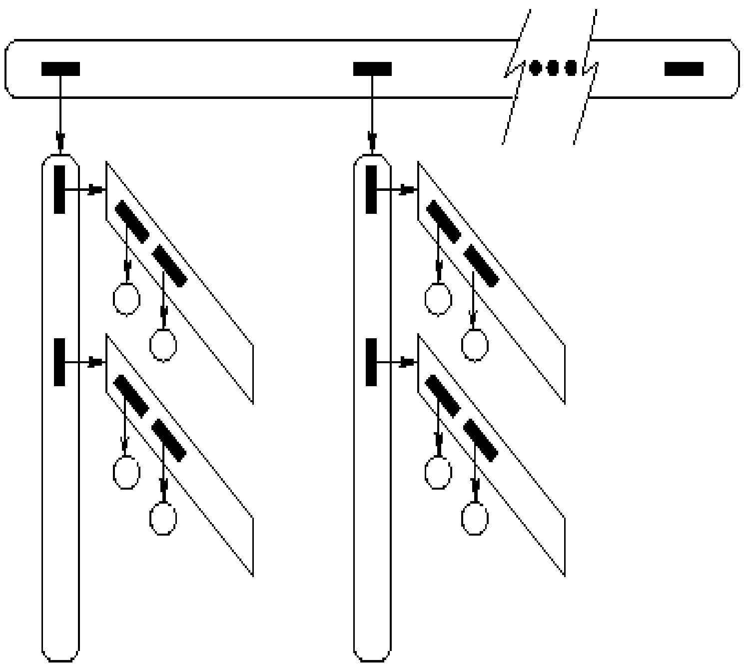
Analyzing Java Arrays

- Traditional loop-level optimizations are not directly applicable to Java arrays
- Multi-dimensional Java arrays may have irregular shapes
  - combness analysis
- Common use of reference (“pointers”),
  - a pointer-based dependence test

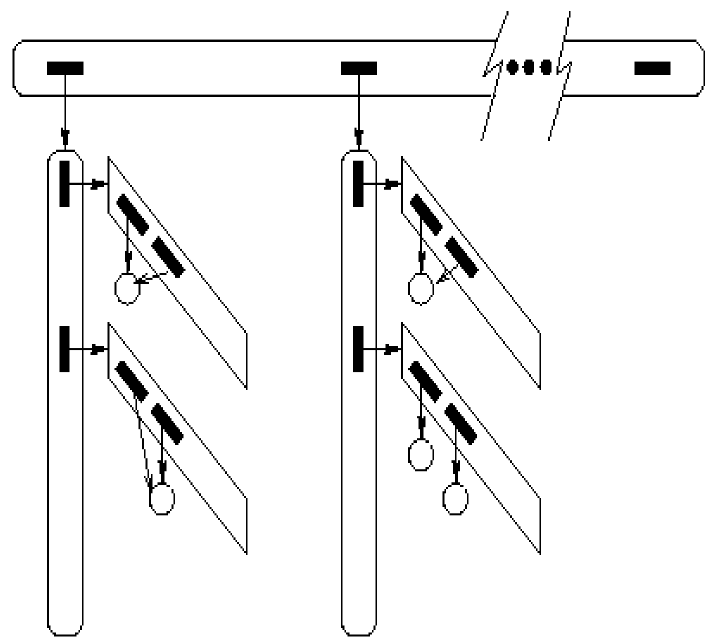

Fortran style optimizations
blocking,
loop unrolling,
loop interchange,
loop fusion,
parallelization,
...
Index-based
DD test
...
alias/shape
exception
Fortran
Java

Combness Analysis

Combness analysis: analyze the shape of an array



level-1, level-2, level-3 comb



level-1, level-2 comb

An Example

- If a is a comb of level-1only, loop-i is parallelizable
- If a is a comb of level-2 only, loop-j is parallelizable
- If a is a comb of level-1 and level-2, the loop-nest-i-j is parallelizable

```
  ...
int a[n][n][n];
...
for ( int i = 0; I<n; I++)
  for (int j = 0; j<n; j++)
     for (int k = 0; k<n; k++)
           a[i][j][k] = a[i][j][k] + 1;
```

Static Performance Prediction and Optimization Control

Joint work with

D. Reed (Illinois) and C. Cascaval (Illinois)

Compile-time Performance Prediction Goals

- Provide the compiler with information to enable performance related optimizations
- Predict the execution time of scientific Fortran programs, with reasonable accuracy, using architecture independent models
- Rapid performance tuning tool
- Architecture evaluation

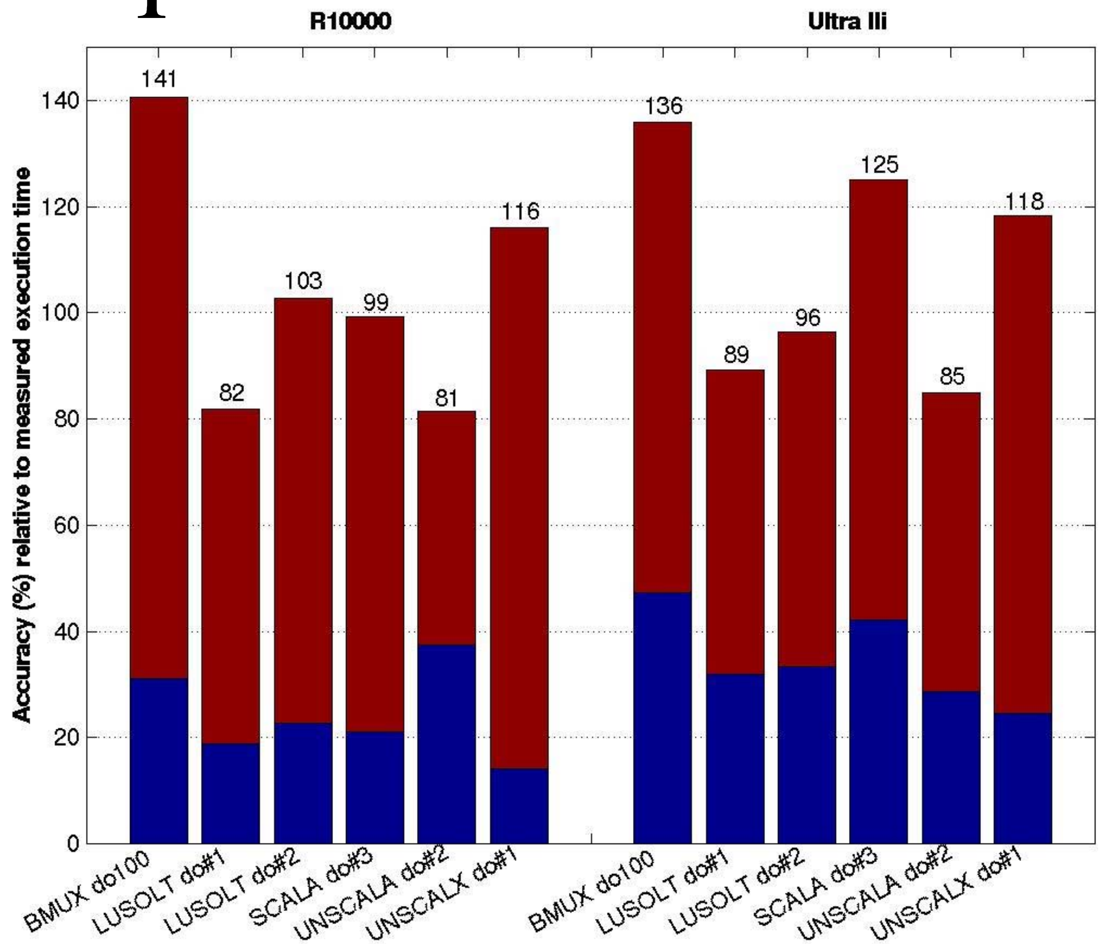
Compile-time Performance Prediction

- Fast performance estimation results with reasonable accuracy
  - light load on the compiler
  - rapid feedback to the user
- Abstract models (symbolic expressions), as a function of
  - program constructs
  - input data set
  - architecture parameters

Performance Prediction Model

- Model different parts of the system independently
- Each model generates one or more terms in a symbolic expression
- Advantages
  - simplicity
  - modularity
  - extensibility

Experimental Results
Splib Execution time



SPECfp95 - Mips R10000