Advanced Topics

Optimization for parallel machines and memory hierarchies

Last Time

- Dependence analysis

Today

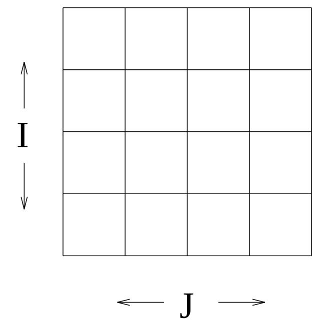
- Loop transformations
- An example - McKinley, Carr, Tseng loop transformations to improve cache performance

Which Loops are Parallel? review

```
   do I = 1, N
     do J = 1, N
S1     A(I,J) = A(I,J-1) + 1

   do I = 1, N
     do J = 1, N
S2     A(I,J) = A(I-1,J-1) + 1

   do I = 1, N
     do J = 1, N
S3     B(I,J) = B(I-1,J+1) + 1
```



- A dependence $D = (d_1, \ldots, d_k)$ is *carried* at *level* i, if d_i is the first nonzero element of the distance/direction vector.
- A loop l_i is *parallel*, if $\nexists$ a dependence D_j carried at level i. Either

	distance vector	direction vector
$\forall D_j$	$d_1, \ldots, d_{i-1} > 0$	$d_1, \ldots, d_{i-1} = $ "<"
OR	$d_1, \ldots, d_i = 0$	$d_1, \ldots, d_i = $ "="

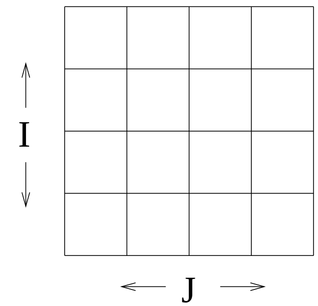
Loop Transformations

Taxonomy

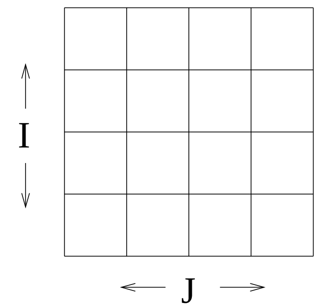
- Loop unrolling
- Loop interchange
- Loop fusion
- Loop distribution (a.k.a. fission)
- Loop skewing
- Strip mine and interchange (a.k.a. tiling & blocking)
- Unroll-and-jam (a variety of tiling)
- Loop reversal

Loop Interchange

```
     do I = 1, N
       do J = 1, N
S1       A(I,J) = A(I-1,J) + 1
       enddo
     enddo
```




```
     do I = 1, N
       do J = 1, N
S2       B(I,J) = B(I-1,J+1) + 1
       enddo
     enddo
```



Loop interchange is safe *iff*

- it does not reverse the execution order of the source and sink of any dependence in the nest, i.e., if the distance vector would become negative.
- ◦ Enables parallelization of outer and/or inner loops
- ◦ Changes execution order of the statements
- ◦ Can improve reuse

Loop Fusion

$\Longrightarrow$ **loop fusion** $\Longrightarrow$

```
     do i = 2, n
s1     a(i) = b(i)

     do i = 2, n
s2     c(i) = b(i) * a(i-1)

     do i = 2, n
s1     a(i) = b(i)
s2     c(i) = b(i) * a(i-1)
```

$\Longleftarrow$ **loop distribution** $\Longleftarrow$

Loop Fusion is safe *iff*

- no forward dependence between nests becomes a backward loop carried dependence.

$\Rightarrow$ Would fusion be safe if s_2 referenced $a(i+1)$?

- Benefits
  - Reuse
  - Eliminates synchronization between parallel loops
  - Reduced loop overhead

Loop Distribution

$\Longrightarrow$ **loop distribution** $\Longrightarrow$

```
     do i = 2, n
s1     a(i) = b(i)
s2     c(i) = b(i) * a(i+1)

     do i = 2, n
s2     c(i) = b(i) * a(i+1)

     do i = 2, n
s1     a(i) = b(i)
```

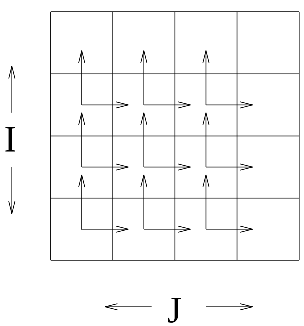
Loop Distribution is safe *iff*

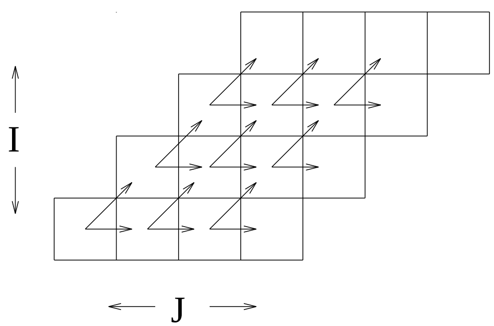
- statements involved in a cycle of dependences (*recurrence*) remain in the same loop, &
- if $\exists$ a dependence between two statements placed in different loops, it must be forward.
- Benefits
  - Partial parallelization
  - Reduces resource requirements
  - Enables other transformations

Loop Skewing and Interchange

```
do I = 1, 100
   do J = 2, 100
      A(I,J) =
         A(I-1,J) + A(I,J-1)
```

```
do I = 1, 100
   do J = I+2, i+100
      A(I,J) =
         A(I-1,J) + A(I,J-1)
```





Loop skewing is always safe.

- The original dependences, $(1,0)$ & $(0,1)$ prevent interchange and parallelization.
- It changes the dependences to $(1,1)$ & $(0,1)$ and after interchange, $(1,1)$ & $(1,0)$, the inner loops is parallel.

$\Rightarrow$ Enables inner loop parallelization

Strip Mine and Interchange

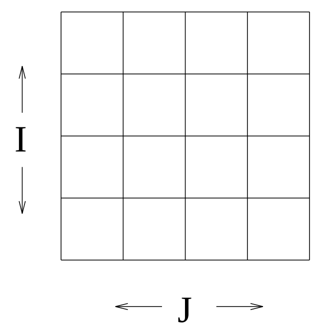
$\Longrightarrow$ Strip Mine $\Longrightarrow$

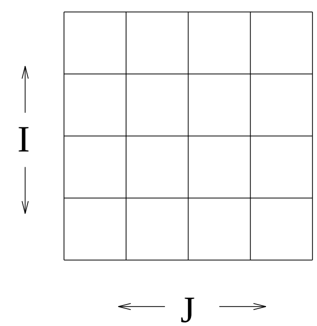
```
do I = 1, n
   do J = 1, n
      A(J,I) = B(J) * C(I)
```

$\Longrightarrow$ Interchange $\Longrightarrow$

```
do II = 1, n, tile
   do I = II, II + tile -1
      do J = 1, n
         A(J,I) = B(J) * C(I)
```

```
do II = 1, n, tile
   do J = 1, n
      do I = II, II + tile -1
         A(J,I) = B(J) * C(I)
```





Strip Mining is always safe. With interchange it

- enables loop invariant reuse by changing the shape of the iteration space

Improving Reuse with Loop Transformations

Motivation: Enable portable programming without sacrificing performance

- optimization framework
- cache model
- compound loop transformation algorithm
  - permutation
  - fusion
  - distribution
  - reversal
- results
  - transformation
  - simulation
  - performance

Optimization Framework

1. improve order of memory accesses to exploit all levels of the memory hierarchy

   $\Longrightarrow$ *cache line size*

2. Tile to fit in cache, second level cache, TLB

   $\Longrightarrow$ *size of cache(s), replacement policy, associativity,*

3. register tiling via unroll-and-jam and scalar replacement

   $\Longrightarrow$ *number and type of registers*

Assumptions

- **cls** - the cache line size in terms of data items
- Fortran column-major order
- interference occurs rarely for small numbers of inner loop iterations

Reuse

- temporal locality — *reuse of a specific location*
- spatial locality — *reuse of adjacent locations (cache lines)*

```
do j = 1, 100
     do k = 1, 100
          do i = 1, 100
               c(i,j) = c(i,j) + a(i,k) * b(k,j)
```

To Determine Temporal and Spatial Reuse:

for each loop l in a nest, consider l innermost

- group references with temporal locality
  $\Rightarrow$ reference groups
- compute the cost in cache lines accessed
  $\Rightarrow$ loop cost
- rank the loops based on their cost
  $\Longrightarrow$ *memory order*

Loop Cost of dmxpy From Linpackd

Loop Cost in cache lines, cls = 4

```
do j = 1, n2
   do i = 1, n1
      y(i)= y(i) + x(j) * m(i,j)
```

reference group	loop i	loop j
y(i)	1/4 n1 $*$ n2	1 $*$ n1
x(j)	1 $*$ n2	1/4 n2 $*$ n1
m(i,j)	1/4 n1 $*$ n2	n2 $*$ n1
total loop cost	1/2 n1 $*$ n2 + n2	5/4 n1 $*$ n2 + n1

Loop permutation

key insight: *If loop l promotes more reuse than loop k at the innermost position, then it will also promote more reuse at any outer position.*

- **memory order**
  (a) is it legal?
  (b) if not find a **nearbyPermutation** $O(n^2)$

$\Longrightarrow$ avoids combinatorial search present in other algorithms

NearbyPermutation

Input:

$\mathcal{O}$ $= \{i_1, i_2, ..., i_n\}$, the original loop ordering
$\mathcal{DV}$ = set of original legal direction vectors for l_n
$\mathcal{L}$ = $\{i_{\sigma_1}, i_{\sigma_2}, \ldots, i_{\sigma_n}\}$, a permutation of $\mathcal{O}$

Output:

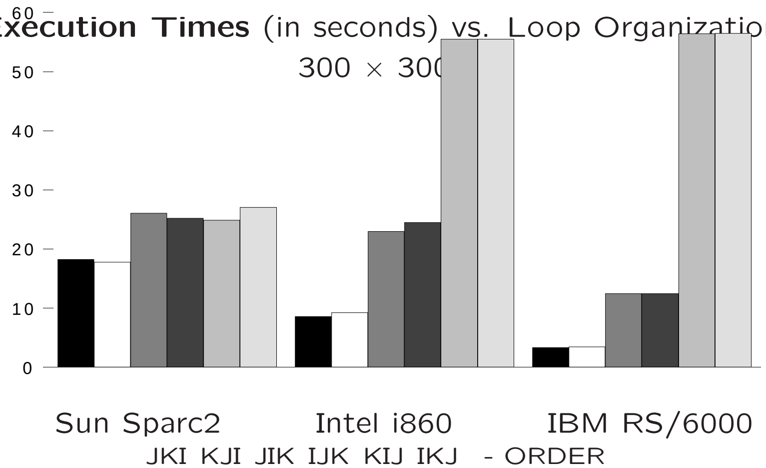
$\mathcal{P}$ a nearby permutation of $\mathcal{O}$

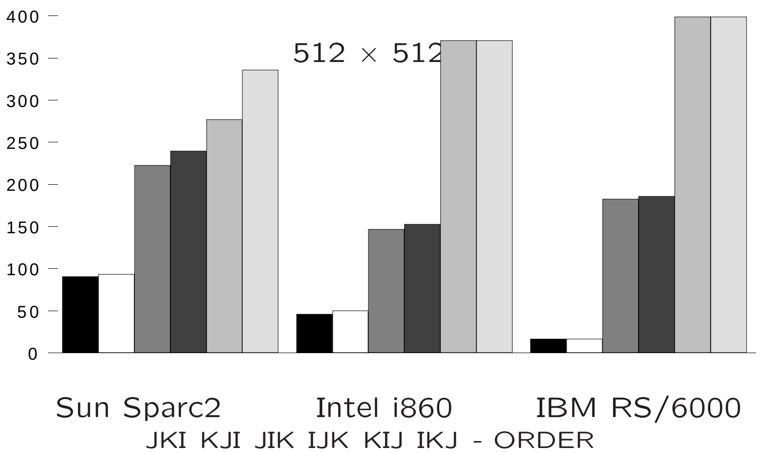
Algorithm:

```
P = ∅ ;   k = 0 ;   m = n
while L ≠ ∅
   for j = 1,m
      l = l_j ∈ L
      if direction vectors for {p_1,...,p_k,l} are legal
         P = {p_1,...,p_k,l}
         L = L − {l} ;   k = k+1 ;   m = m−1
         break for
      endif
   endfor
endwhile
```

Matrix Multiply - execution times in seconds

Execution Times (in seconds) vs. Loop Organization





Loop Fusion Erlebacher

Fortran 90 loops for ADI Integration

```
DO I = 2, N
   X(I,1:N) = X(I,1:N) - X(I-1,1:N)*A(I,1:N)/B(I-1,1:N)
   B(I,1:N) = B(I,1:N) - A(I,1:N)*A(I,1:N)/B(I-1,1:N)
```

⇓ *simple translation to Fortran 77*

```
DO I = 2, N
   DO K = 1, N
      X(I,K) = X(I,K) - X(I-1,K)*A(I,K)/B(I-1,K)
   DO K = 1, N
      B(I,K) = B(I,K) - A(I,K)*A(I,K)/B(I-1,K)
```

⇓ *loop fusion & interchange*

```
DO K = 1, N
   DO I = 2, N
      X(I,K) = X(I,K) - X(I-1,K)*A(I,K)/B(I-1,K)
      B(I,K) = B(I,K) - A(I,K)*A(I,K)/B(I-1,K)
```

Execution times in seconds.

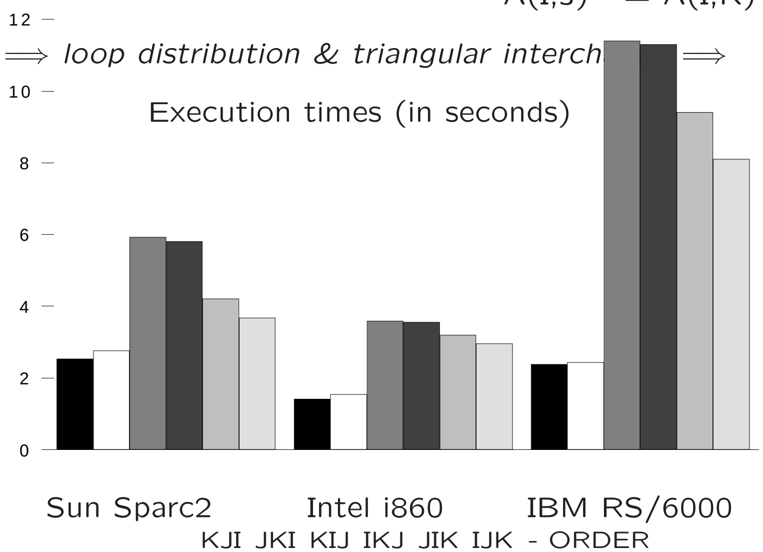
		Memory Order	
Processor	Original	Distributed	Fused
Sun Sparc2	.806	.813	.672
Intel i860	.547	.548	.518
IBM RS6000	.390	.400	.383

Loop Distribution   Cholesky Factorization

```
DO K = 1,N
   A(K,K) = SQRT(A(K,K))
   DO I = K+1,N
      A(I,K) = A(I,K)/A(K,K)
      DO J = K+1,I
         A(I,J) -= A(I,K)*A(J,K)
```

```
DO K = 1,N
  A(K,K) = SQRT(A(K,K))
  DO I = K+1,N
    A(I,K) = A(I,K)/A(K,K)
  DO J = K,N
    DO I=J+1,N
      A(I,J) -= A(I,K)*A(J,K)
```

$\Longrightarrow$ *loop distribution & triangular interch* $\Longrightarrow$



Algorithm Summary

for each nest $\mathcal{L}_|$ in a set of adjacent nests

- compute reference groups for each l_i
- compute loop cost for each l_i and sort
- permutation with reversal?
- fuse inner loops and permute?
- distribute and permute?

fuse nests $\mathcal{L}_|$?

Results

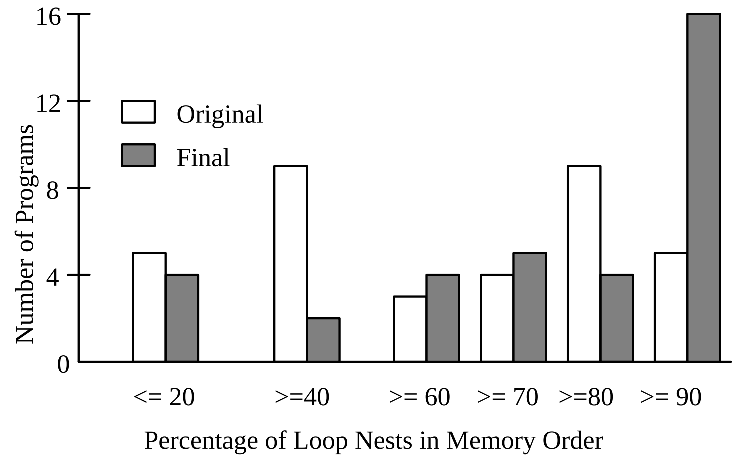
test suite

- Perfect Benchmarks
- SPEC Benchmarks
- NAS Benchmarks
- 4 additional programs

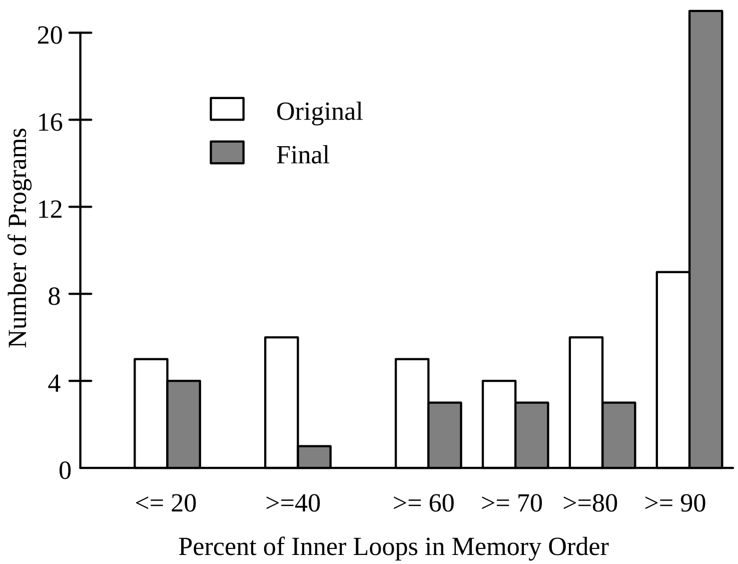
experiments

- ◦ on our ability to transform programs
- ◦ simulated hit rates for RS/6000 and i860
- ◦ execution times on an RS/6000

Achieving Memory Order for Loop Nests



Achieving Memory Order for Inner Loops



Simulated Cache Hit Rates

cache1:	RS/6000	64K cache, 4-way, 128 byte cache line
cache2:	i860	8K cache, 2-way, 32 byte cache line

for RS/6000

$\Longrightarrow$ in 12 of 27 programs, optimized procedures started with 100% hit rates

$\Longrightarrow$ 98.86 – average hit rate for original programs

Perfect Club	Memory Order			Opt. Proc. Hit Rates			
	Org	Prm	Fail	Cache 1		Cache 2	
	%			org	final	org	final
adm	52	16	32	100	100	**97.7**	**97.8**
arc2d	55	28	17	**89.0**	**98.5**	**68.3**	**91.9**
bdna	75	18	7	100	100	100	100
dyfesm	63	15	22	100	100	100	100
flo52	83	17	0	99.6	99.6	**96.7**	**96.3**
mdg	83	8	8	100	100	87.4	87.4
mg3d	95	3	3	**98.8**	**99.7**	**95.3**	**98.7**
ocean	82	13	5	100	100	**93.0**	**92.8**
qcd	53	11	36	100	100	100	100
spec77	64	7	29	100	100	100	100
track	50	16	34	100	100	100	100
trfd	52	0	48	99.9	99.9	93.7	93.7

Performance Results in Seconds on RS6000

Program	Original	Transformed	Speedup
arc2d	410.13	190.69	2.15
dyfesm	25.42	25.37	1.00
flo52	62.06	61.62	1.01
dnasa7 (btrix)	36.18	30.27	1.20
dnasa7 (emit)	16.46	16.39	1.00
dnasa7 (gmtry)	155.30	17.89	8.68
dnasa7(vpenta)	149.68	115.62	1.29
applu	146.61	149.49	0.98
appsp	361.43	337.84	1.07
linpackd	159.04	157.48	1.01
simple	963.20	850.18	1.13
wave	445.94	414.60	1.08

Summary

Recap of Transformation Results

- 80 % of nests were permuted into memory order
- 85 % of inner loops were permuted into memory order
- loop permutation is the most effective optimization
- 229 candidates for fusion, resulting in 80 nests
- 23 nests were distributed, resulting in 52 nests

Observations

- many programs started out with high hit ratios
- smaller cache sizes result in higher improvements in hit rates

$\Longrightarrow$ regardless of the original target architecture, compiler optimizations improve locality for uniprocessors

Next Time

Compiling for an EDGE Arcuitecture

Read: D. Burger et al., Compiling for TRIPS: Scaling to the End of Silicon with EDGE Architectures, IEEE Computer, pp. 44-55, July 2004.