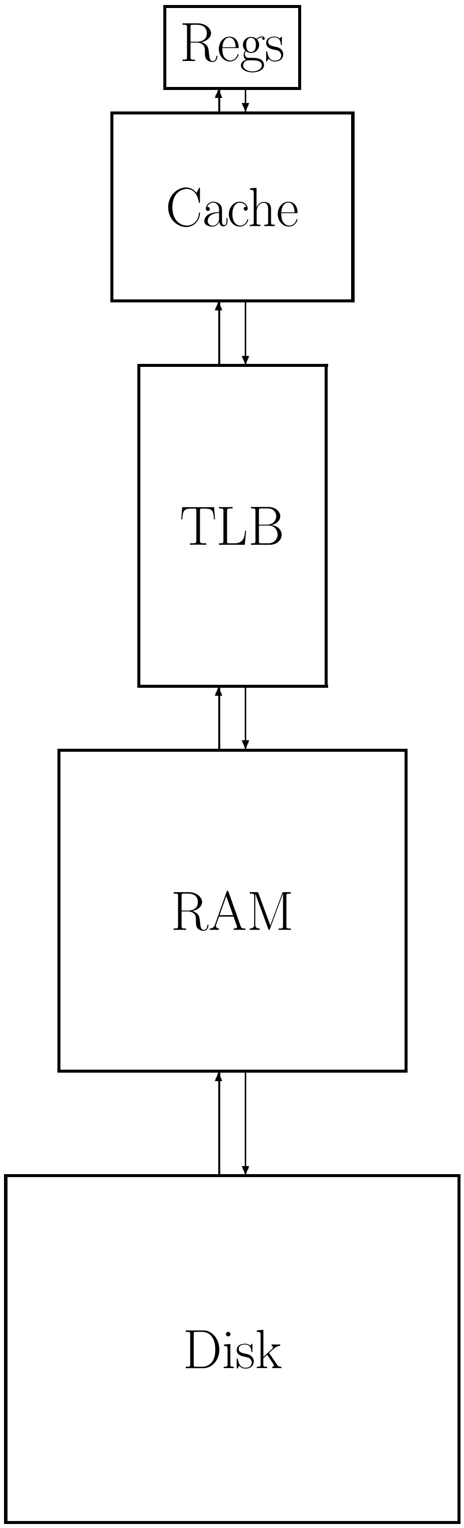
Optimizations for memory hierachies

- Carr, McKinley, Tseng
  loop transformations to improve cache performance
- Callahan, Carr, Kennedy
  transformation to improve register allocation
  (scalar replacement)

Memory Hierarchy — sample architecture



32 registers
$\times$ 8 bytes/register

512 lines
$\times$ 128 bytes/line

128 pages entries
$\times$ 72 bytes/page

8192 pages
$\times$ 4096 bytes/page

131072 tracks
$\times$ 8192 bytes/track

Data Locality

Why locality?

- memory accesses are expensive
- exploit higher levels of memory hierarchy by reusing registers, cache lines, TLB, etc.
- locality of reference $\Leftrightarrow$ reuse

Locality

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

Reuse

- self-reuse — *caused by same reference*
- group-reuse — *caused by multiple references*

What locality/reuse occurs in this loop nest?

```
do i = 1, N
  do j = 1, N
      A(i) = A(i) + B(j) + B(j+2)
```

M. Wolf and M. Lam, "A Data Locality Optimizing Algorithm," SIGPLAN '91 Conference on Progsramming Language Design and Implementation

Loop Transformations

What?

- modify execution order of loop iterations
- preserve data dependence constraints

Why?

- data locality – increase reuse of registers, cache
- parallelism – eliminate loop–carried deps, incr. granularity

Taxonomy

- Loop Interchange∗
- Loop Fusion∗
- Loop Distribution∗
- Strip Mine and Interchange (a.k.a. Tiling & Blocking)
- Unroll-and-Jam (a variety of Tiling)
- Loop Reversal∗

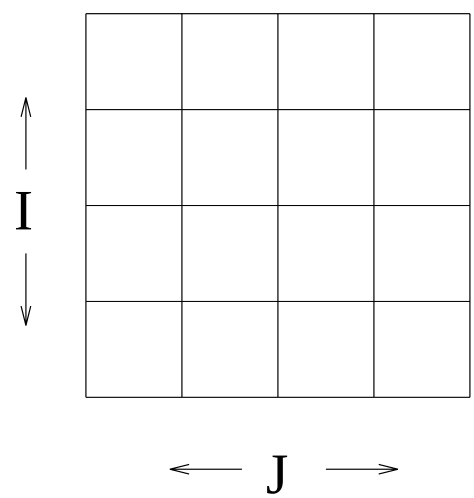
∗: used in *compound* algorithm

Review — Which Loops are Parallel?

```
    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
```

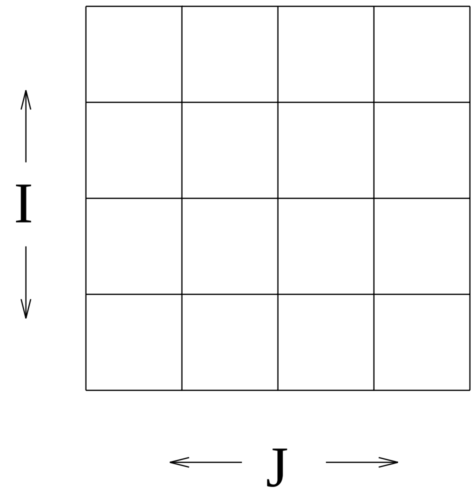


- A dependence $D = (d_1, \ldots, d_k)$ is *carried* at *level* i, if d_i is the first nonzero element of the distance/direction vector.

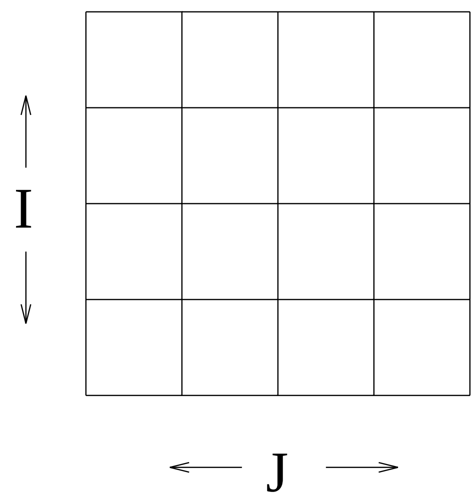
- A loop l_i is *parallel*, if $\nexists$ a dependence D carried at level i. Either

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.

$\Rightarrow$ Benefits

- Enable parallelization of outer or inner loops
- Changes execution order of the statements
- May improve reuse

Loop Fusion

$$\Longrightarrow \textbf{loop fusion} \Longrightarrow$$

```
    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)
```

$$\Longleftarrow \textbf{loop distribution} \Longleftarrow$$

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
  - May improve reuse
  - Eliminates synchronization between parallel loops
  - Reduced loop overhead

Loop Distribution (loop fission)

$\Longrightarrow$ 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.

$\Rightarrow$ Benefits

- ◦ Partial parallelization
- ◦ Enables other transformations (e.g. loop interchange)

Strip Mine and Interchange

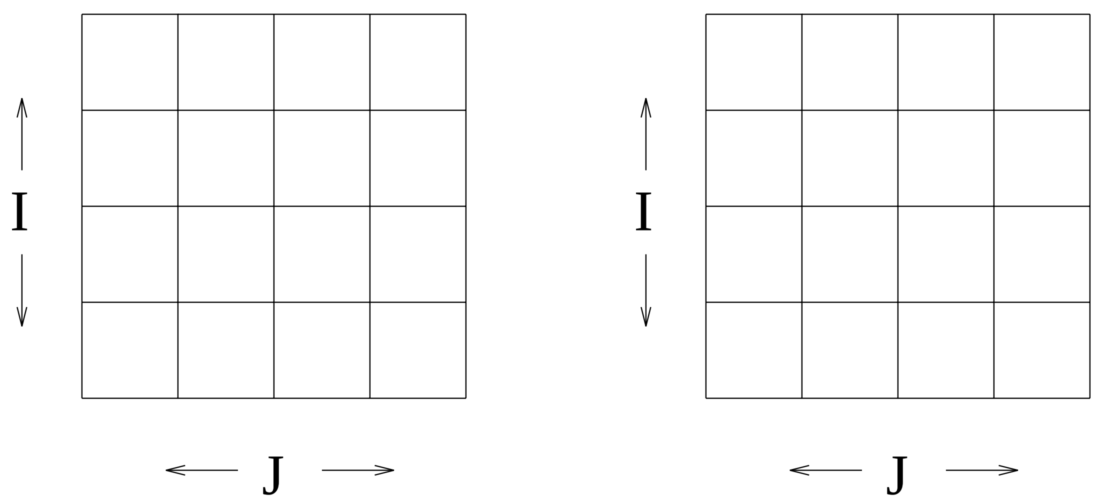
$\Longrightarrow$ Strip Mine $\Longrightarrow$

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

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

$\Longrightarrow$ Interchange $\Longrightarrow$

```
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

Using Loop Transformations Systematically to Improve Reuse

Motivation: Enable portable programming without sacrificing performance

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

K. S. McKinley, S. Carr & C.W. Tseng, "Improving Data Locality with Loop Transformations", *ACM Transactions on Programming Languages and Systems*, Vol. 18, No.4, July 1996.

Optimization Framework

Data locality optimizations should proceed in the following order:

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

   $\Longrightarrow$ *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*

Step 1: Assumptions (mostly machine independent)

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

Loop Transformations to Improve Reuse

To Determine *Temporal* and *Spatial* Reuse:

for each loop l in a nest, consider l innermost

- partition references with group reuse (temporal and spatial locality)
  $\Rightarrow$ reference groups
- compute the cost in cache lines accessed
  $\Rightarrow$ loop cost
- rank the loops based on their cost
  $\Longrightarrow$ *memory order* is loop order with minimal cost

Key insight

If loop l promotes more reuse than loop k at the innermost position, then it probably promotes more reuse at any outer position

Selecting a loop permutation

- select *memory order* if legal
- if not, find a nearby legal permutation
- avoids evaluating many permutations

Reference Groups

Goal: Avoid overcounting cache lines accessed by multiple references that most likely access the same set of cache lines.

Two references Ref_1 and Ref_2 are in the same reference group *with respect to loop l* if:

1. (Group–temporal reuse)
   $\exists \quad Ref_1 \; \delta \; Ref_2$ (including input dep.) and
   (a) δ is a loop–independent dependence, or
   (b) δ_l is a small constant $d (\leq 2)$, and all other entries are 0, or
2. (Group–spatial reuse)
   Ref_1 and Ref_2 refer to the same array and differ by at most d' in the first subscript dimension ($d' \leq cls$). All other subscripts must be identical.

Reference Groups – Example

```
do k = 2, N-1
   do j = 2, N-1
     do i = 2, N-1
       A(i,j,k) = A(i+1,j+1,k) + B(i,j,k) +
                  B(i,j+1,k) + B(i+1,j,k)
```

Reference Groups – Example

```
do k = 2, N-1
   do j = 2, N-1
     do i = 2, N-1
       A(i,j,k) = A(i+1,j+1,k) + B(i,j,k) +
                  B(i,j+1,k) + B(i+1,j,k)
```

for **loop j**:	for **loops i & k**
{ A(i,j,k) }	{ A(i,j,k) }
{ A(i+1,j+1,k) }	{ A(i+1,j+1,k) }
{ B(i,j,k), B(i,j+1,k), B(i+1,j,k) }	{ B(i,j,k), B(i+1,j,k) }
	{ B(i,j+1,k) }

Selecting a Loop Permutation

Cost of reference group for loop k

1. select representative from reference group
2. find cost (in cache lines) with k innermost

invariant	1
unit-stride	$(U_k - L_k + 1)/cls$
otherwise	$U_k - L_k + 1$

3. multiply by trip counts of outer loops

Loop cost $=$ sum of costs for reference groups

Matrix multiplication example

```
do j = 1, N
   do k = 1, N
     do i = 1, N
       C(i,j) = C(i,j) + A(i,k) * B(k,j)
```

RefGroups	J	K	I
C(i,j)	$n * n^2$	$1 * n^2$	$\frac{1}{4}n * n^2$
A(i,k)	$1 * n^2$	$n * n^2$	$\frac{1}{4}n * n^2$
B(k,j)	$n * n^2$	$\frac{1}{4}n * n^2$	$1 * n^2$
total	$2n^3 + n^2$	$\frac{5}{4}n^3 + n^2$	$\frac{1}{2}n^3 + n^2$

LoopCost $(with\ cls = 4)$

NearbyPermutation

INPUT:

$\mathcal{O} \quad = \{i_1, i_2, ..., i_n\}$, the original loop ordering

$\mathcal{DV} = $ set of original legal direction vectors for l_n

$\mathcal{L} \quad = \{i_{\sigma_1}, i_{\sigma_2}, \ldots, i_{\sigma_n}\}$, a permutation of $\mathcal{O}$

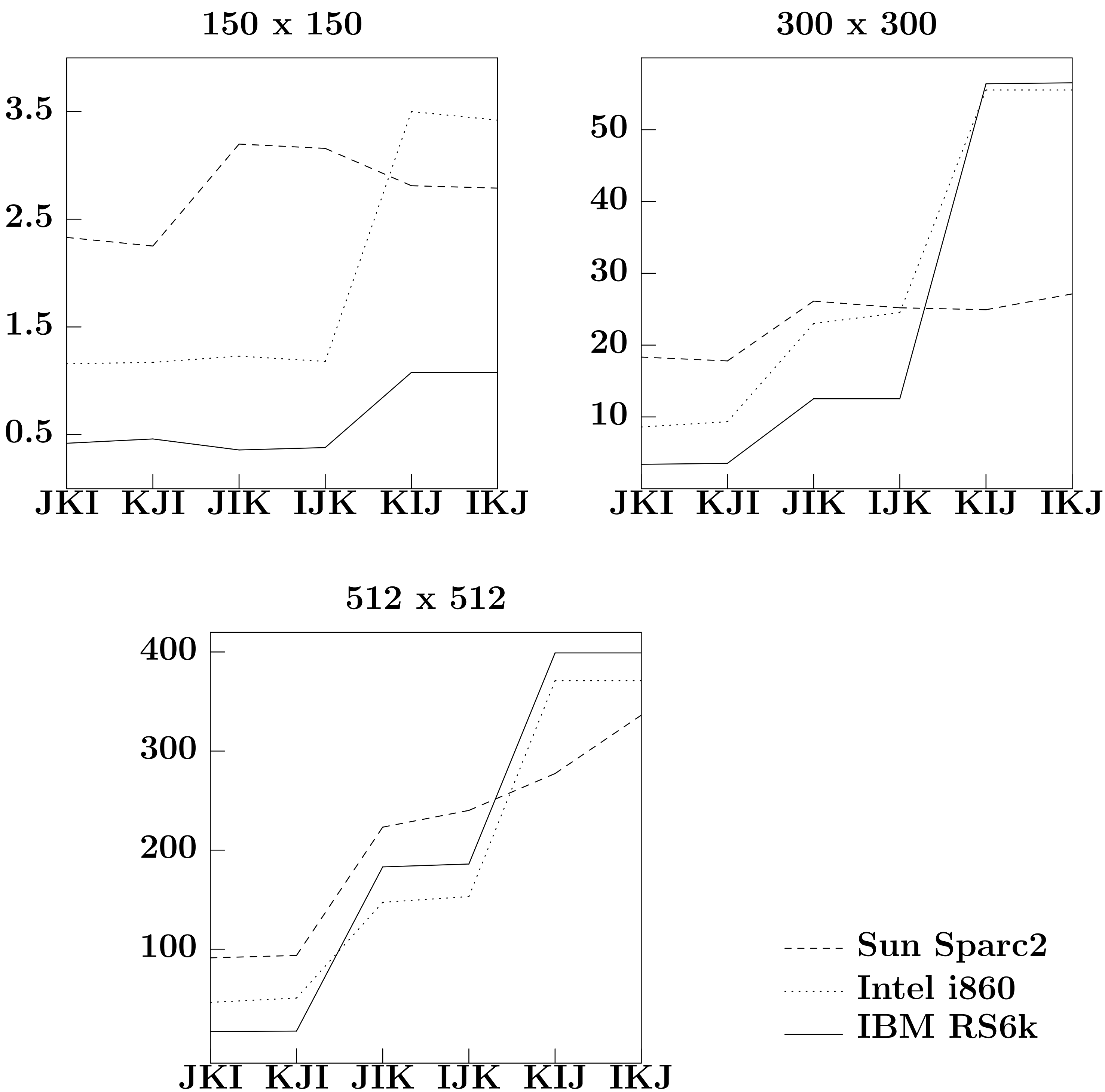
OUTPUT:

$\mathcal{P}$ a nearby permutation of $\mathcal{O}$ as close to $\mathcal{L}$ as possible

ALGORITHM:

$\mathcal{P} = \emptyset \; ; \quad k = 0 \; ; \quad m = n$
while $\mathcal{L} \neq \emptyset$
  for $j = 1, m$
    $l = l_j \in \mathcal{L}$
    if direction vectors for $\{p_1, \ldots, p_k, l\}$ are legal
      $\mathcal{P} = \{p_1, \ldots, p_k, l\}$
      $\mathcal{L} = \mathcal{L} - \{l\} \; ; \quad k = k + 1 \; ; \quad m = m - 1$
      break for
    endif
  endfor
endwhile

Matrix Multiply - execution times in seconds



Loop Fusion

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)
```

$\Downarrow$ *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)
```

$\Downarrow$ *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)
```

Example: Erlebacher - ADI integration program written in a Fortran 90 style

Loop Fusion

Two goals:

- improve temporal locality
- fuse all inner loops, creating a nest that is permutable

`Distributed` — hand distributed and put into memory order

- ◦ degrades locality between loop nests
- ◦ increases locality within loop nests

`Fused` — fusion only done if profitable

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

Fusion is always an improvement (up to 17%).

Algorithm Summary

Goal: minimize actual *LoopCost* by achieving memory order for as many statements in the nest as possible.

for each nest L_j 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 L_j?

Implementation:

- on top of ParaScope
- 25% increase in compilation time over just parsing and dependence analysis
- 33% increase over dependence analysis

Results

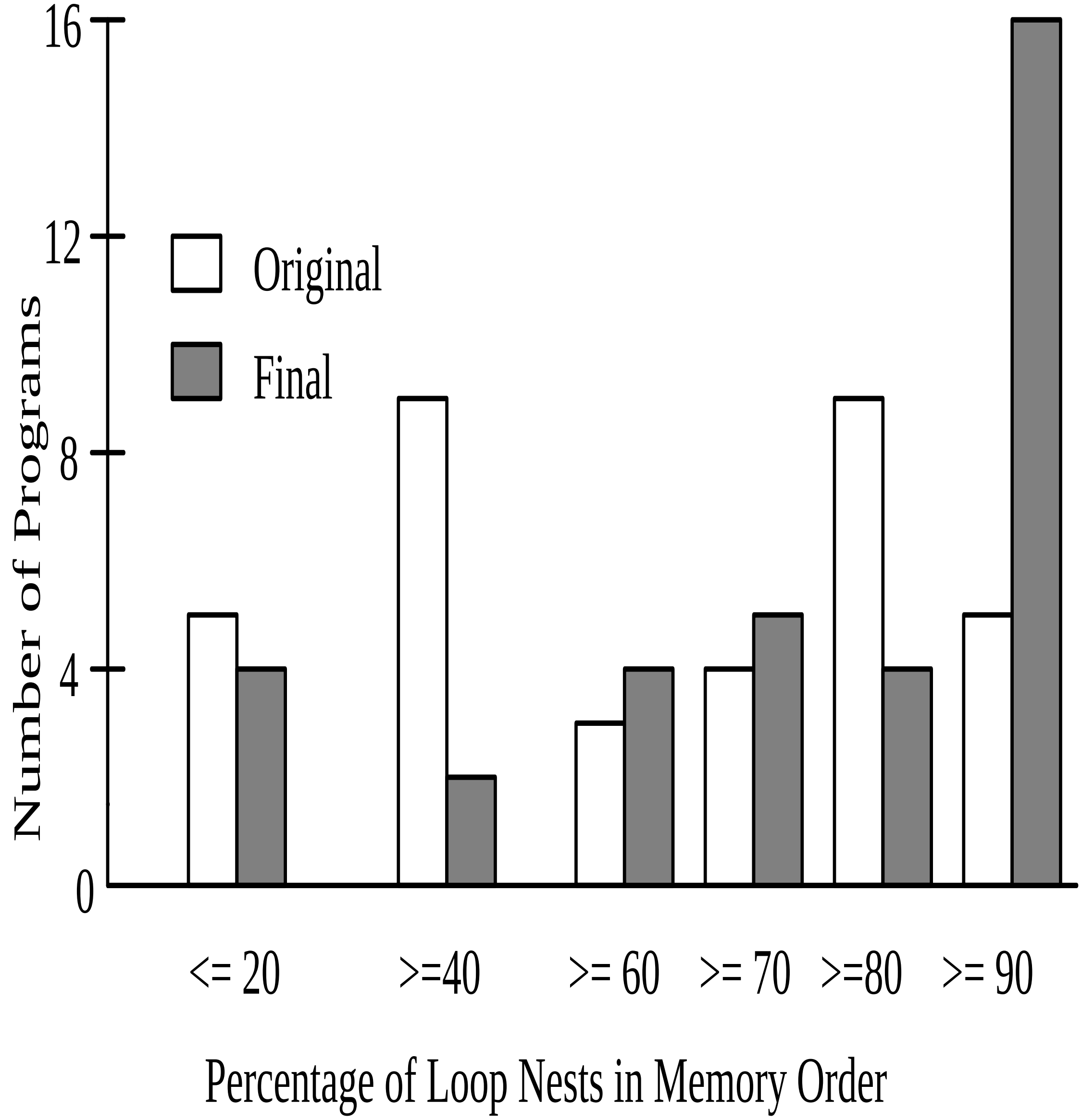
test suite (35 programs)

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

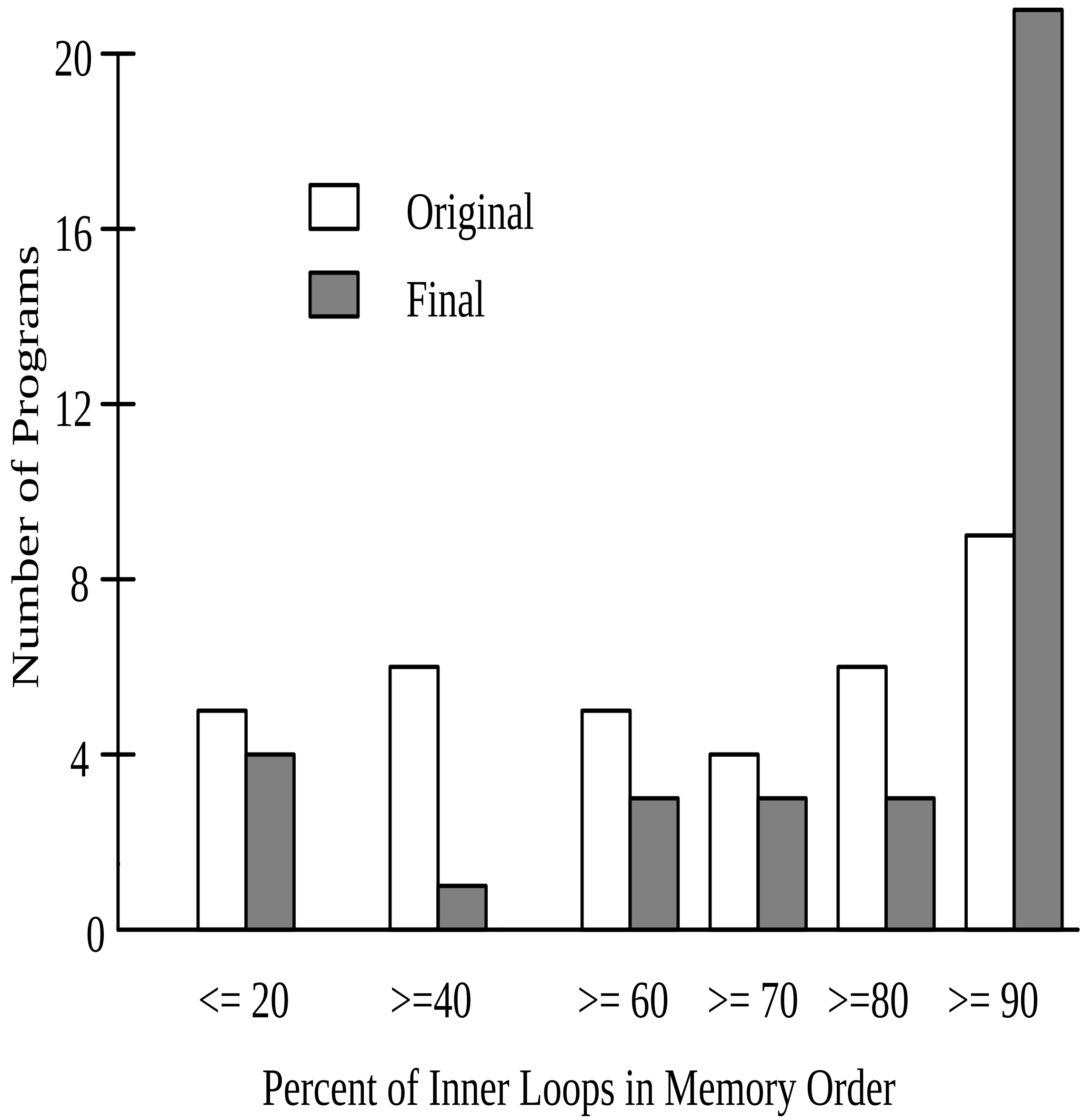
experiments

- on 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



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

Scalar Replacement

Problem: register allocators never keep *a(i)* in a register
Idea: trick the allocator

1. locate patterns of consistent re-use
2. replace load with a copy into temporary
3. replace store with copy from temporary
4. may need copies at end of loop (re-use spans > 1 iteration)

Benefits

- decrease number of loads and stores
- keep re-used values in registers
- often see improvements by factors of $2\times$ to $3\times$

Carr, "Memory-Hierarchy Management," Dissertation, Rice University, September 1992.

Scalar Replacement

```
do i = 1, n                    do i = 1, n
   do j = 1, n                    t = a(i)
      a(i) = a(i) + b(j)          do j = 1, n
   enddo                             t = t + b(j)
enddo                             enddo
                                  a(i) = t
                               enddo
```

Scalar replacement exposes the reuse of `a(i)`

- traditional scalar analysis is inadequate
- use dependence analysis to understand array references

```
do i = 1, n                 t = a(i - 1)
   a(i) = a(i - 1)          do i = 1, n
enddo                          a(i) = t
                               t = a(i)
                            enddo
```