Optimizations for memory hierarchies

- Carr, McKinley, Tseng
  loop transformations to improve cache performance

- Callahan, Carr, Kennedy
  transformation to improve register allocation
  (scalar replacement)
Memory Hierarchy — sample architecture

- **Regs**: 32 registers
  - × 8 bytes/register
- **Cache**: 512 lines
  - × 128 bytes/line
- **TLB**: 128 pages entries
  - × 72 bytes/page
- **RAM**: 8192 pages
  - × 4096 bytes/page
- **Disk**: 131072 tracks
  - × 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 ⇔ reuse

Locality

- temporal locality  \textit{reuse of a specific location}
- spatial locality  \textit{reuse of adjacent locations}  
  \textit{(cache lines, TLB entries, pages)}

Reuse

- self-reuse  \textit{caused by same reference}
- group-reuse  \textit{caused by multiple references}

What locality/reuse occurs in this loop nest?

\begin{verbatim}
    do i = 1, N
        do j = 1, N
            A(i) = A(i) + B(j) + B(j+2)
    \end{verbatim}

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?

\[
\begin{align*}
S_1 & \quad A(I,J) = A(I,J-1) + 1 \\
S_2 & \quad A(I,J) = A(I-1,J-1) + 1 \\
S_3 & \quad B(I,J) = B(I-1,J+1) + 1
\end{align*}
\]

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

- A loop \( l_i \) is \textit{parallel}, if \( \not\exists \) a dependence \( D \) carried at level \( i \). Either
Loop Interchange

do I = 1, N
  do J = 1, N
    \( S_1 \) \( A(I,J) = A(I-1,J) + 1 \)
    enddo
  enddo

\( S_2 \) \( B(I,J) = B(I-1,J+1) + 1 \)
  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.

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

\[ \begin{align*}
\text{do } i &= 2, n \\
\text{\hspace{1cm}} s_1 &\ a(i) = b(i) \\
\text{\hspace{1cm}} s_2 &\ c(i) = b(i) \ast a(i-1)
\end{align*} \]

\[ \text{\hspace{1cm}} \text{do } i &= 2, n \\
\text{\hspace{2cm}} s_1 &\ a(i) = b(i) \\
\text{\hspace{2cm}} s_2 &\ c(i) = b(i) \ast a(i-1)
\]

\[ \implies \text{loop fusion} \implies \]

\[ \iff \text{loop distribution} \iff \]

Loop Fusion is safe \textit{iff}

\begin{itemize}
  \item no forward dependence between nests becomes a backward loop carried dependence.
\end{itemize}

\[ \implies \text{Would fusion be safe if } s_2 \text{ referenced } a(i + 1) \ ? \]

\begin{itemize}
  \item Benefits
    \begin{itemize}
      \item May improve reuse
      \item Eliminates synchronization between parallel loops
      \item Reduced loop overhead
    \end{itemize}
\end{itemize}
Loop Distribution (loop fission)

\[ \Rightarrow \text{loop distribution} \Rightarrow \]
\[
\text{do } i = 2, n \\
\quad s_1 \quad a(i) = b(i) \\
\quad s_2 \quad c(i) = b(i) \ast a(i+1)
\]
\[
\text{do } i = 2, n \\
\quad s_2 \quad c(i) = b(i) \ast a(i+1)
\]
\[
\text{do } i = 2, n \\
\quad s_1 \quad a(i) = b(i)
\]

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.

\[ \Rightarrow \text{Benefits} \]
- Partial parallelization
- Enables other transformations (e.g. loop interchange)
Strip Mine and Interchange

⇒ Strip Mine ⇒

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)

⇒ Interchange ⇒

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

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

   \[ \implies \text{cache line size} \]

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

   \[ \implies \text{size of cache(s), replacement policy, associativity} \]

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

   \[ \implies \text{number and type of registers} \]

Step 1: Assumptions (mostly machine independent)

- \text{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
  \( \Rightarrow \) 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) $\delta$ is a loop–independent dependence, or
   (b) $\delta_l$ 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

dk = 2, N-1
   dj = 2, N-1
      di = 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)

<table>
<thead>
<tr>
<th>for loop j:</th>
<th>for loops i &amp; k</th>
</tr>
</thead>
<tbody>
<tr>
<td>{ A(i,j,k) }</td>
<td>{ A(i,j,k) }</td>
</tr>
<tr>
<td>{ A(i+1,j+1,k) }</td>
<td>{ A(i+1,j+1,k) }</td>
</tr>
<tr>
<td>{ B(i,j,k), B(i,j+1,k), B(i+1,j,k) }</td>
<td>{ B(i,j,k), B(i+1,j,k), B(i+1,j+1,k) }</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>invariant</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>unit-stride</td>
<td>((U_k - L_k + 1)/\text{cls})</td>
</tr>
<tr>
<td>otherwise</td>
<td>(U_k - L_k + 1)</td>
</tr>
</tbody>
</table>

3. multiply by trip counts of outer loops

Loop cost = sum of costs for reference groups

Matrix multiplication example

\[
\begin{align*}
\text{do } & j = 1, N \\
& \text{do } k = 1, N \\
& \quad \text{do } i = 1, N \\
& \quad \quad C(i,j) = C(i,j) + A(i,k) \times B(k,j)
\end{align*}
\]

<table>
<thead>
<tr>
<th>RefGroups</th>
<th>J</th>
<th>K</th>
<th>I</th>
</tr>
</thead>
<tbody>
<tr>
<td>C(i,j)</td>
<td>(n \times n^2)</td>
<td>(1 \times n^2)</td>
<td>(\frac{1}{4}n \times n^2)</td>
</tr>
<tr>
<td>A(i,k)</td>
<td>(1 \times n^2)</td>
<td>(n \times n^2)</td>
<td>(\frac{1}{4}n \times n^2)</td>
</tr>
<tr>
<td>B(k,j)</td>
<td>(n \times n^2)</td>
<td>(\frac{1}{4}n \times n^2)</td>
<td>(1 \times n^2)</td>
</tr>
<tr>
<td>total</td>
<td>(2n^3 + n^2)</td>
<td>(\frac{5}{4}n^3 + n^2)</td>
<td>(\frac{1}{2}n^3 + n^2)</td>
</tr>
</tbody>
</table>

LoopCost (with \( \text{cls} = 4 \))
NearbyPermutation

**Input:**
- $O = \{i_1, i_2, ..., i_n\}$, the original loop ordering
- $D\mathcal{V}$ = set of original legal direction vectors for $l_n$
- $\mathcal{L} = \{i_{\sigma_1}, i_{\sigma_2}, ..., i_{\sigma_n}\}$, a permutation of $O$

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

**Algorithm:**
\[
P = \emptyset ; \quad k = 0 ; \quad m = n \\
\text{while } \mathcal{L} \neq \emptyset \\
\quad \text{for } j = 1, m \\
\quad \quad l = l_j \in \mathcal{L} \\
\quad \quad \text{if direction vectors for } \{p_1, ..., p_k, l\} \text{ are legal} \\
\quad \quad \quad P = \{p_1, ..., p_k, l\} \\
\quad \quad \quad \mathcal{L} = \mathcal{L} - \{l\} ; \quad k = k + 1 ; \quad m = m - 1 \\
\quad \quad \quad \text{break for} \\
\quad \quad \text{endif} \\
\quad \text{endfor} \\
\text{endwhile}
\]
Matrix Multiply - execution times in seconds

150 x 150

300 x 300

512 x 512

--- Sun Sparc2
----- Intel i860
------ IBM RS6k
Loop Fusion

*Fortran 90 loops for ADI Integration*

```fortran
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)
END DO
```

```fortran
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)
   END DO
   B(I,K) = B(I,K) - A(I,K)*A(I,K)/B(I-1,K)
END DO
```

```fortran
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)
   END DO
```

**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

<table>
<thead>
<tr>
<th>Processor</th>
<th>Original</th>
<th>Memory Order</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Distributed</td>
</tr>
<tr>
<td>Sun Sparc2</td>
<td>.806</td>
<td>.813</td>
</tr>
<tr>
<td>Intel i860</td>
<td>.547</td>
<td>.548</td>
</tr>
<tr>
<td>IBM RS6000</td>
<td>.390</td>
<td>.400</td>
</tr>
</tbody>
</table>

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

![Bar Chart]

- **Original**
- **Final**

<table>
<thead>
<tr>
<th>Percentage of Loop Nests in Memory Order</th>
<th>Original</th>
<th>Final</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;= 20</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt;= 40</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt;= 60</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt;= 70</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt;= 80</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&gt;= 90</td>
<td></td>
<td>16</td>
</tr>
</tbody>
</table>
Achieving Memory Order for Inner Loops

Percent of Inner Loops in Memory Order

Number of Programs

Original
Final

<= 20  >= 40  >= 60  >= 70  >= 80  >= 90

>= 40           >= 60    >= 70   >= 80    >= 90
Performance Results in Seconds on RS6000

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

⇒ 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 \((\text{re-use spans} > 1 \text{ iteration})\)

Benefits

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

Scalar Replacement

\[
\begin{align*}
\text{do } i &= 1, n \\
\text{do } j &= 1, n \\
a(i) &= a(i) + b(j) \\
\text{enddo} \\
\text{enddo}
\end{align*}
\]

\[
\begin{align*}
\text{do } i &= 1, n \\
t &= a(i) \\
\text{do } j &= 1, n \\
t &= t + b(j) \\
\text{enddo} \\
a(i) &= t \\
\text{enddo}
\end{align*}
\]

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

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

\[
\begin{align*}
\text{do } i &= 1, n \\
a(i) &= a(i - 1) \\
\text{enddo} \\
t &= a(i - 1) \\
\text{do } i &= 1, n \\
a(i) &= t \\
t &= a(i) \\
\text{enddo}
\end{align*}
\]