Cache Management

Improving Memory Locality and
Reducing Memory Latency

Introduction

- Memory system performance is critical in modern architectures
  - Accessing memory takes much longer than accessing cache
- Optimizations
  - Reuse data already in cache (locality)
    - Reduce memory bandwidth requirement
  - Prefetch data ahead of time
    - Reduce memory latency requirement
- Two types of cache reuse
  - Temporal reuse
    - After bringing a value into cache, use the same value multiple times
  - Spatial reuse
    - After bringing a value into cache, use its neighboring values in the same cache line
- Cache reuse is limited by
  - cache size, cache line size, cache associativity, replacement policy

```
DO I = 1, M
  DO J = 1, N
    A(I) = A(I) + B(J)
  ENDDO
ENDDO
```

```
DO I = 1, M
  DO J = 1, N
    A(I, J)=A(I,J)+B(I,J)
  ENDDO
ENDDO
```

Optimizing Memory Performance

- Improve cache reuse
  - Loop interchange
  - Loop blocking (strip-mining + interchange)
  - Loop blocking + skewing
- Reduce memory latency
  - Software prefetching

Loop Interchange

- Which loop should be innermost ?
  - Reduce the number of interfering data accesses between reuse of the same (or neighboring) data
- Approach: attach a cost function when each loop is placed innermost
  - Assuming cache line size is L

```
DO I = 1, N
  DO J = 1, N
    DO K = 1, N
      C(I, J) = C(I, J) + A(I, K) * B(K, J)
    ENDDO
  ENDDO
ENDDO
```

  - Innermost K loop = N*N*N*(1+1/L)+N*N
  - Innermost J loop = 2*N*N*N+N*N
  - Innermost I loop = 2*N*N*N/L+N*N
- Reorder loop from innermost in the order of increasing cost
  - Limited by safety of loop interchange

Loop Blocking

- Goal: separate computation into blocks, where cache can hold the entire data used by each block
- Example

```
DO J = 1, M
   DO I = 1, N
      D(I) = D(I) + B(I,J)
   ENDDO
ENDDO
```

  - Assuming N is large, 2*N*M/C cache misses (memory accesses)

- After blocking (**strip-mine-and-interchange)**

```
DO jj = 1, M, T
   DO I = 1, N
      DO J = jj, MIN(jj+T-1, M)
         D(I) = D(I) + B(I, J)
      ENDDO
   ENDDO
ENDDO
```

  - Assuming T is small, (**M/T)*(N/C) + M*N/C** misses

Alternative Ways of Blocking

```
DO jj = 1, M, T
   DO I = 1, N
      DO J = jj, MIN(jj+T-1, M)
         D(I) = D(I) + B(I, J)
      ENDDO
   ENDDO
ENDDO
```

```
DO ii = 1, N, T
   DO J = 1, M
      DO I = ii, MIN(ii+T-1, N)
         D(I) = D(I) + B(I, J)
      ENDDO
   ENDDO
ENDDO
```

```
DO jj = 1, M, Tj
   DO ii = 1, N, Ti
      DO J = jj, MIN(jj+Tj-1,M)
         DO I = ii, MIN(ii+Ti-1, N)
             D(I) = D(I) + B(I, J)
         ENDDO
      ENDDO
   ENDDO
ENDDO
```

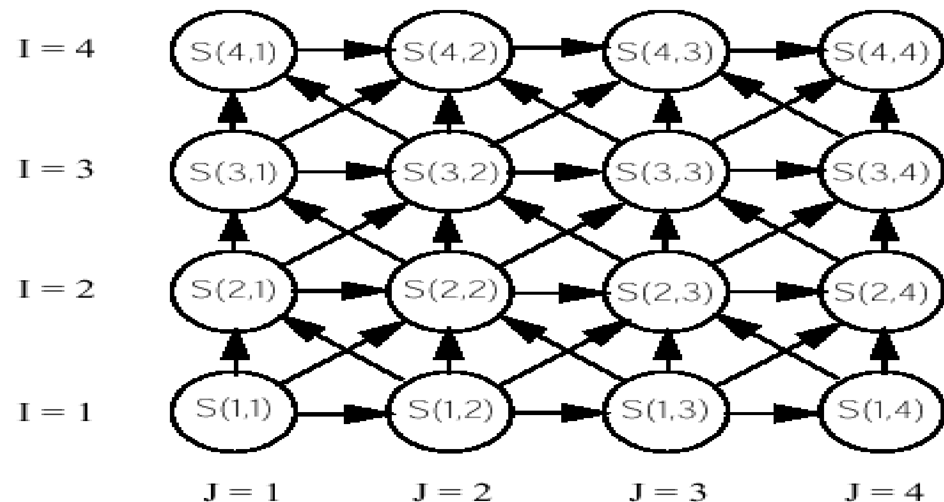
The Blocking Transformation

- The transformation takes a group of loops L0,…,Lk
  - Strip-mine each loop Li into two loops Li′ and Li′′
  - Move all strip counting loops L0′,L1′,…,Lk′ to the outside
  - Leave all strip traversing loops L0′′,L1′′,…,L,′′ inside
- Safety of blocking
  - Strip-mining is always legal
  - Loop interchange is not always legal
  - All participating loops must be safe to be moved outside
    - Each loop has only "=" or "<" in all dependence vectors
- Profitability of Blocking: can enable cache reuse by an outer loop that
  - Carries small-threshold dependences (including input dep)
  - The loop index appears (with small stride) in the contiguous dimension of an array and in no other dimension

Blocking with Skewing

- Goal: enable loop interchange that is not legal otherwise

```
DO I = 1, M
   DO J = 1, N
      A(J+1) =
          (A(J)+A(J+1))/2
   ENDDO
ENDDO
```

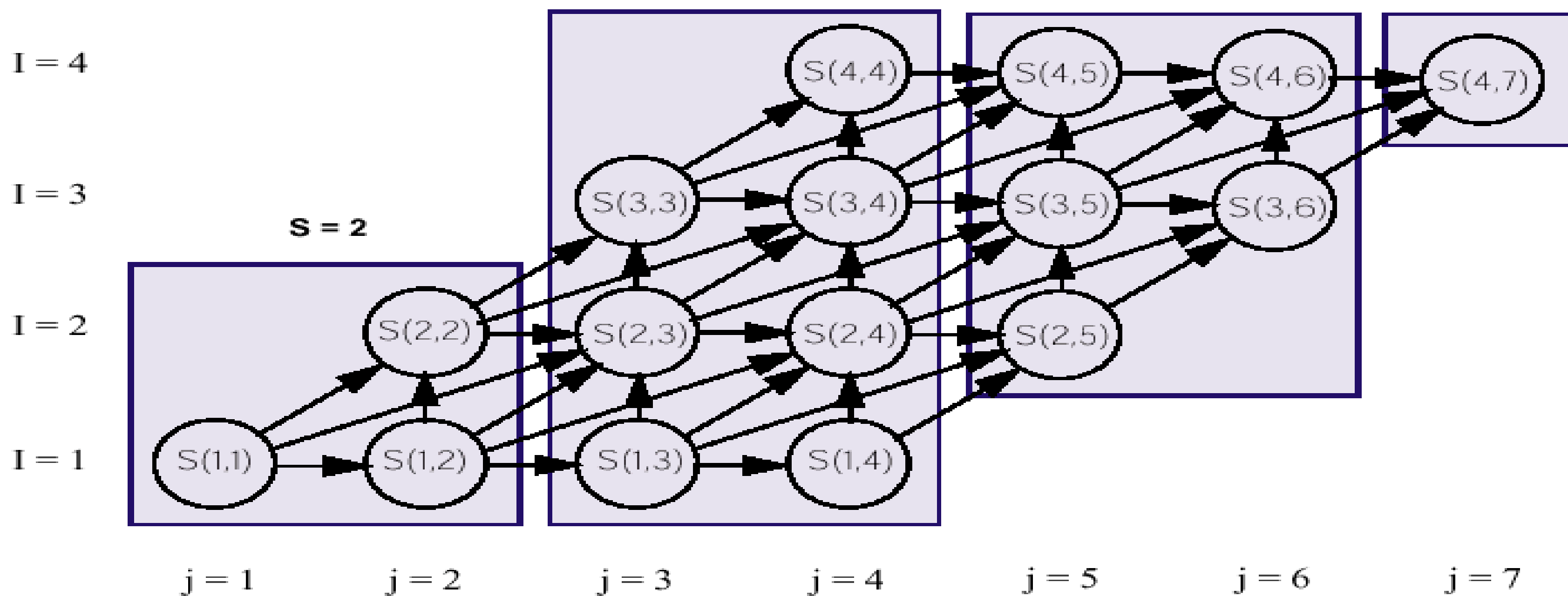


- After skewing

```
DO I = 1, N
   DO j = I, M+I-1
       A(j-I+2) = (A(j-I+1) + A(j-I+2))/2
   ENDDO
ENDDO
```

Blocking with Skewing

```
DO jj = 1, M+N-1, S
   DO I = MAX(1, j-M+1), MIN(j, N)
      DO J = jj, MAX(jj+S-1, M+I-1)
           A(J-I+2) = (A(J-I+1)+A(J-I+2))/2
```



Triangular Blocking

Input code

```
DO I = 2, N
  DO J = 1, I-1
    A(I, J) = A(I, I) + A(J, J)
  ENDDO
ENDDO
```

After strip-mining

```
DO ii = 2, N, T
  DO I = ii, MIN(ii+T-1,N)
    DO J = 1, I – 1
      A(I, J) = A(I,I)+A(I,J)
    ENDDO
  ENDDO
ENDDO
```

After interchange

```
DO ii = 2, N, T
  DO J = 1, MIN(ii+T-2,N-1)
    DO I = MAX(J+1, ii), MIN(ii+T-1,N)
      A(I, J) = A(I, I) + A(I, J)
    ENDDO
  ENDDO
ENDDO
```

Software Prefetching

- Goal: prefetch data known to be used in the near future
  - Support by hardware: discard prefetch if already in cache
- Safety: never alter the meaning of program
- Profitability: can reduce memory access latency if none of the following happens
  - Other useful data are evicted from cache due to the operation
  - The prefetched data are evicted before use or never used
- Critical steps in an effective prefetching algorithm
  - Accurately determine which references to prefetch
  - Insert the prefetch op just far enough in advance

Prefetch Analysis

- Assume loop nests have been blocked for locality
- Identify where cache misses may happen
  - Eliminate dependences unlikely to result in cache reuse
  - For each loop that carries reuse
    - Estimate size of data accessed by each loop iteration
    - Determine # of iterations where data would overflow cache
    - Any dependence with a threshold equal to or greater than the overflow is considered ineffective for reuse
- Partition memory references into groups
  - Each group has a generator that brings data to cache
  - All other references in each group can reuse data in cache
- Identify where prefetching is required
  - Is the group generator contained in a dep cycle carried by the loop?
    - If no, a miss is expected on each iteration, or every CL iterations where CL is the cache line size
    - If yes, a miss is expected only on the first few accesses, depending on the distance of the carrying dependence

Prefetch Analysis Example

```
DO J = 1, M
  DO I = 1, N
    A(I, J) = A(I, J) + C(J) + B(I)
  ENDDO
ENDDO
```

- Data volume by x iterations of each loop
  - loopI: 2*x+1 overflow iteration: x=(CS-CL+1)/2
  - loopJ: 2*N*x+x overflow iteration: x=CS/(2*N+CL)
- Reference groups
  - A(I,J): a miss every CL iterations of loopI
  - B(I): a miss every CL iterations of loopI
  - C(J): a miss every CL iterations of loopJ

Inserting Prefetch for Acyclic Reference Groups

```
DO J = 1, M
   DO I = 1, N
      A(I, J) = A(I, J) + C(J)
   ENDDO
ENDDO
```

```
DO J = 1, M
   prefetch(A(1,J))
   DO I = 1, 3
      A(I, J) = A(I, J) + C(J)
   ENDDO
   DO ii = 4, M, 4
      prefetch(A(ii, J))
      DO I = ii, MIN(M,ii+4)
         A(I, J) = A(I, J) + C(J)
      ENDDO
   ENDDO
ENDDO
```

- The reference group
  - A(I,J): a miss every CL iterations of loopI
  - Assuming CL=4, then i0 = 5 and Ti = 4

Inserting Prefetch Operations for Acyclic Reference Groups

- If there is no spatial reuse of the reference
  - insert a prefetch before reference to the group generator
- If the references have spatial locality
  - Let i0 = the first loop iteration where reference to the group generator is regularly a cache miss
  - Let Ti = the interval of loop iterations for cache miss
  - Partition the loop into two parts;
    - initial subloop running from 1 to i0-1 and
    - remainder running from i0 to the end
  - Strip-mine the remainder loop with step Ti
  - Insert prefetch operations to avoid misses
  - Eliminate any very short loops by unrolling

Inserting Prefetch for Cyclic Reference Groups

- Insert prefetch prior to the loop carrying the dependence cycle
- If an outer loop L carries the dependence, insert a prefetch loop
  - If the innermost prefetch loop gets data in unit stride, split it into
    - A prefetch of the first group generator reference
    - Remaider loop strip-mined to prefetch the next cache line at every iteration

```
Prefetch B(1)
DO I=4,M,4
   prefetch(B(I))
ENDDO
DO jj = 1,M,4
  prefetch(C(jj))
  DO J=jj,MIN(M,jj+3)
   DO ii = 1, M, 4
     prefetch(A(ii, J))
     DO I = ii, MIN(M,ii+4)
        A(I, J) = A(I, J)+C(J)+B(I)
     ENDDO
   ENDDO
  ENDDO
ENDDO
```

Prefetch Irregular Accesses

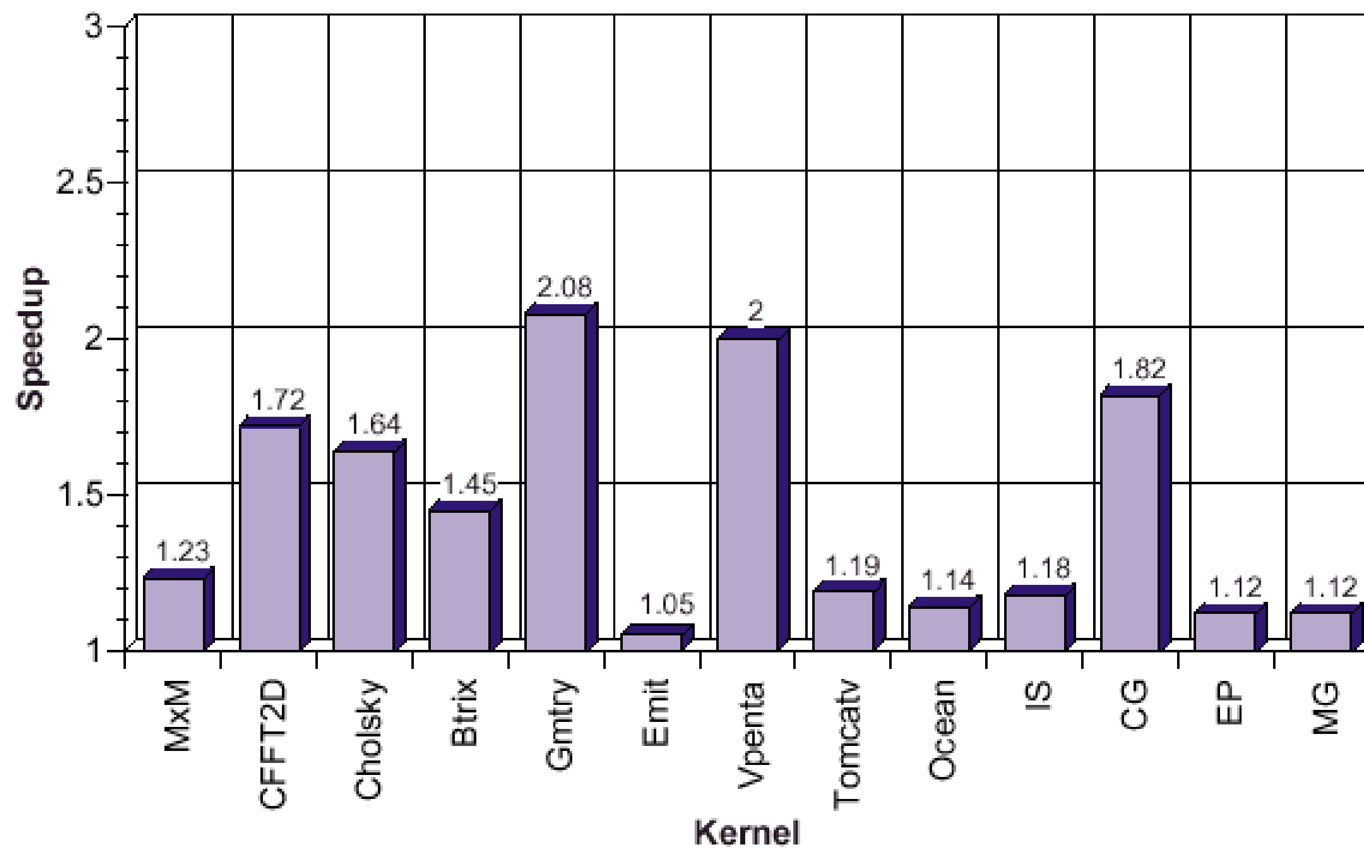
- Input code

```
DO J = 1, M
  DO I = 2, 33
    A(I, J) = A(I, J) * B(IX(I), J)
  ENDDO
ENDDO
```

- After prefetch transformation

```
prefetch(IX(2))
DO I = 5, 33, 4
  prefetch(IX(I))
ENDDO
……
```

Effectiveness of Software Prefetching



Summary

- Two different kind of cache reuse
  - Temporal reuse
  - Spatial reuse
- Strategies to increase cache reuse
  - Loop interchange
  - Loop blocking (strip-mining + interchange)
  - Loop blocking + skewing
- Software prefetching: reduce memory latency
  - Works only when the memory bandwidth is not saturated