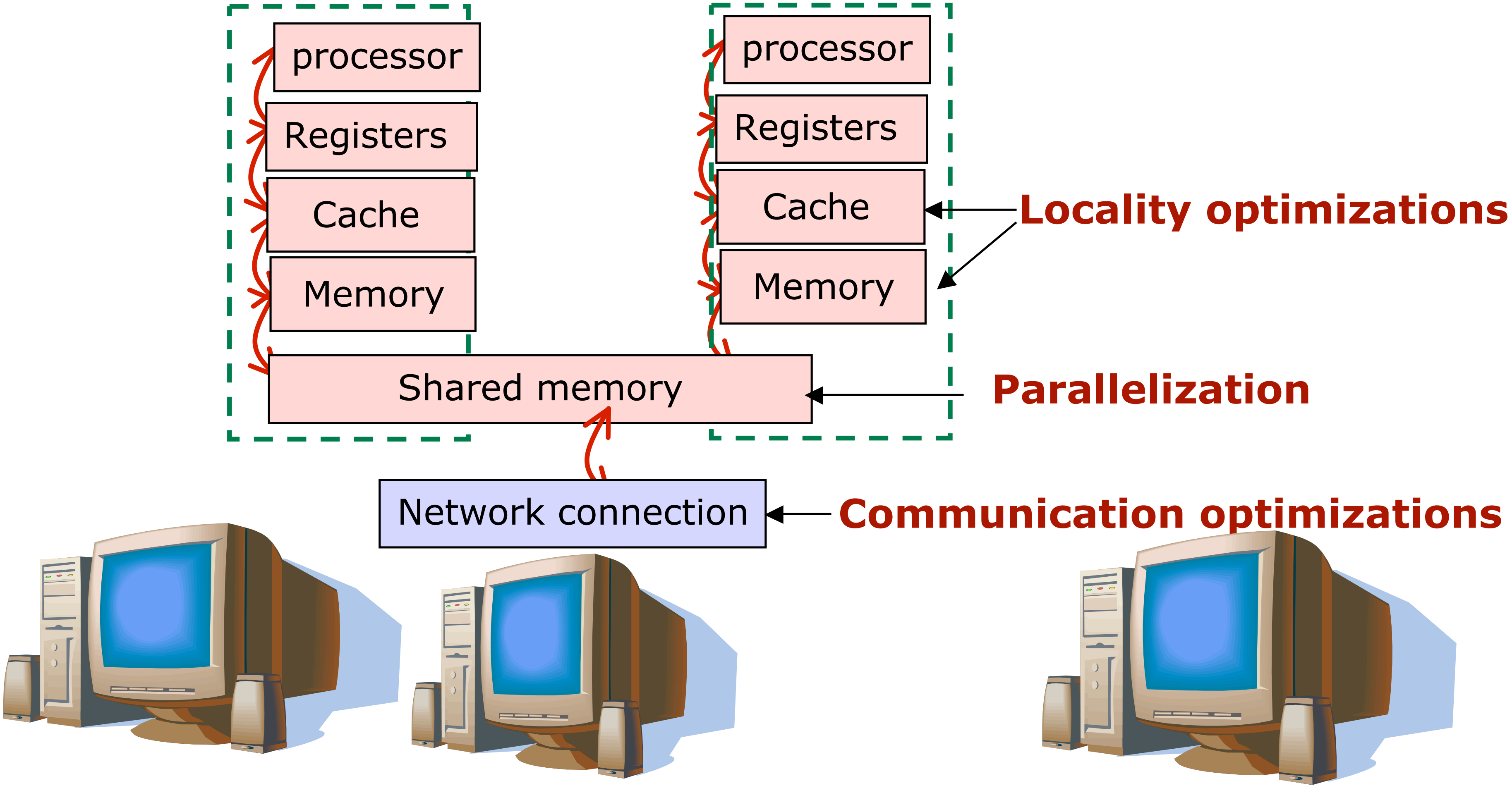
Improving Data Layout and Register Usage

Scalar Replacement and array copying, loop unroll-and-jam

Optimizing for Register Usage

- Registers are part of the memory hierarchy
  - Compare to cache, compilers have complete control over what data to put in register
  - Can use registers to hold scalar variables
- Goal: convert array references to scalars
  - Dynamically determine what data to put in registers
    - Compare to dynamic data layout transformations
- Optimizations to improve register usage
  - Scalar Replacement and array copying
    - Unified as dynamic memory layout optimizations
    - Yi LCPC05: Applying Data Copy To Improve Memory Performance of General Array Computations
  - Unroll-and-Jam
  - Other transformations
    - loop interchange, fusion

Improving Memory Performance



processor

Registers

Cache

Memory

processor

Registers

Cache

Memory

Shared memory

Network connection

Locality optimizations

Parallelization

Communication optimizations

Compiler Optimizations For Locality

- Computation optimizations
  - Loop blocking, fusion, unroll-and-jam, interchange, unrolling
    - Rearrange computations for better spatial and temporal locality
- Data-layout optimizations: rearrange layout of data (arrays, linked data structures)
  - Static layout transformation
    - A single layout for global variables throughout the entire application
    - No additional overhead, tradeoff between different layout choices
  - Dynamic layout transformation
    - Dynamically determine how to lay out data for each computation phase
    - Flexible but could be expensive
- Combining computation and data transformations
  - Static layout transformation
    - Transform layout first, then computation
  - Dynamic layout transformation
    - Transform computation first, then dynamically re-arrange layout

Scalar Replacement

- Convert array references to scalar variables to improve performance of register allocation

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

- **A(I) can be left in a register throughout the inner loop**

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

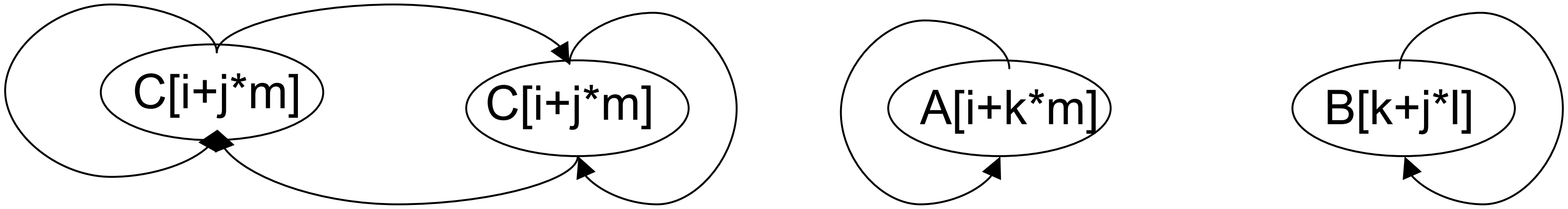
- **All loads and stores to A in the inner loop have been eliminated**
- **High chance of T being allocated to a register by register allocation**

Array Copying vs. Scalar Replacement

- Array copying: dynamic layout transformation for arrays
  - Copy arrays into local buffers before computation
  - Copy modified local buffers back to array
- Previous work
  - Lam, Rothberg and Wolf, Temam, Granston and Jalby
    - Copy arrays after loop blocking
  - Optimizing irregular applications
    - Data access patterns not known until runtime
    - Dynamic layout transformation --- through libraries
  - Scalar Replacement
    - Equivalent to copying single array element into scalars
    - Carr and Kennedy: applied to inner loops
- Unify scalar replacement and array copying (Yi LCPC'05)
  - Improve cache and register locality
  - Automaticly insert copy operations to ensure safety
  - Heuristics to reduce buffer size and copy cost

Array Copy: Matrix Multiplication

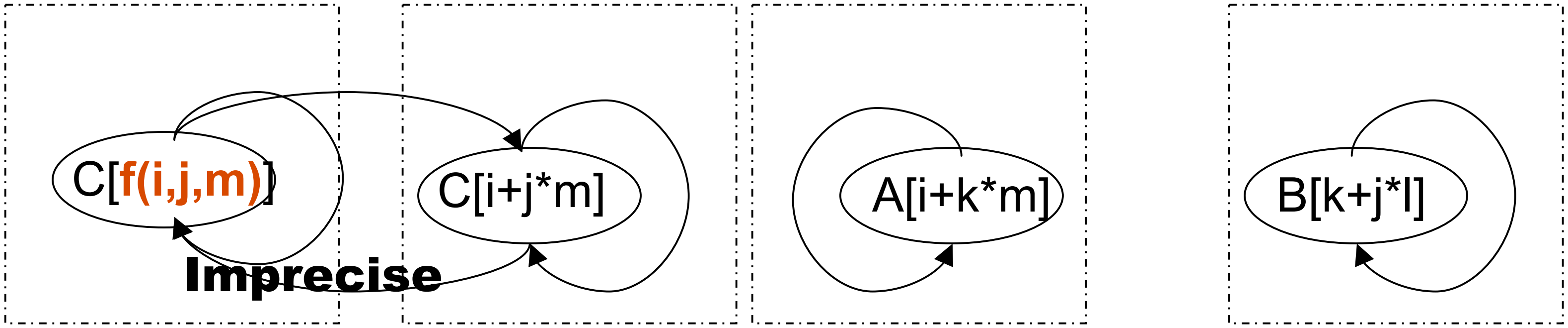
```
for (j=0; j<n; ++j)
   for (k=0; k<l; ++k)
    for (i=0; i<m; ++i)
     C[i+j*m] = C[i+j*m] + alpha * A[i+k*m]*B[k+j*l];
```



- Step1: build dependence graph
  - True, output, anti and input deps between array references
  - Is each dependence consistent/precise?
    - Dependences with constant distance?
    - src and sink always refer to the same memory store?

Array Copy: Imprecise Dependences

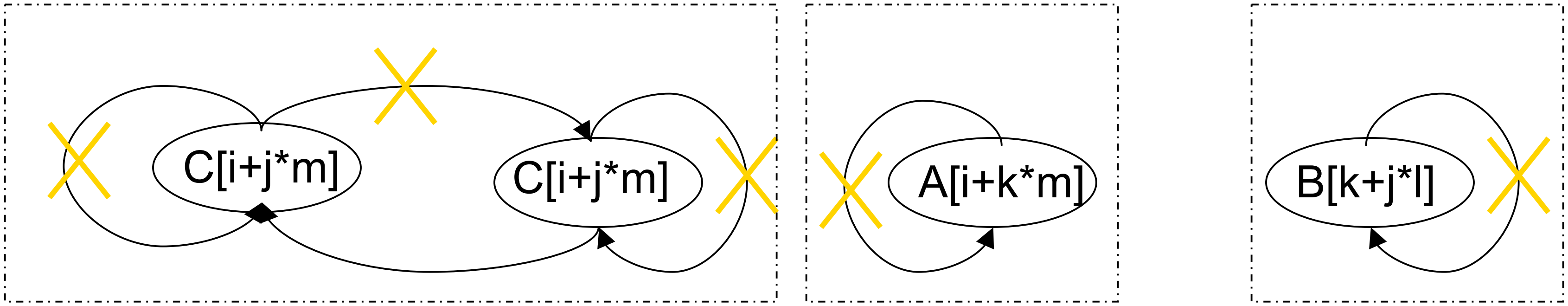
```
for (j=0; j<n; ++j)
  for (k=0; k<l; ++k)
   for (i=0; i<m; ++i) {
    C[f(i,j,m)]= C[i+j*m] + alpha * A[i+k*m]*B[k+j*l];
```



- Array references connected by imprecise deps
  - Cannot precisely determine a mapping between subscripts
  - Sometimes may refer to the same location, sometimes not
  - Not safe to copy into a single buffer
    - Never attempt to copy them

Safety of Applying Array Copy

```
for (j=0; j<n; ++j)
   for (k=0; k<l; ++k)
    for (i=0; i<m; ++i)
     C[i+j*m] = C[i+j*m] + alpha * A[i+k*m]*B[k+j*l];
```



- Identify arrays connected by imprecise deps (cannot be copied)
  - All dependences are precise in matrix multiplication
- Remove all cycles in the dependence graph
  - Impose an order on all array references
    - C[i+j*m] (R) -> A[i+k*m] -> B[k+j*l] -> C[i+j*m] (W)
  - Remove all back edges (no more cycles in dep graph)
- Apply typed fusion to group array references that can be copied

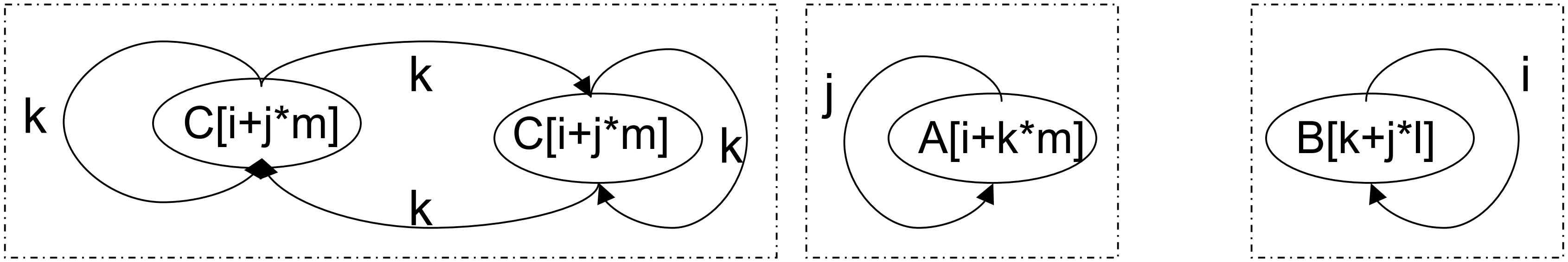
Profitability of Applying Array Copy

```
for (j=0; j<n; ++j)
   for (k=0; k<l; ++k)
    for (i=0; i<m; ++i)
     C[i+j*m] = C[i+j*m] + alpha * A[i+k*m]*B[k+j*l];
```

location to copy A

location to copy C

location to copy B



- Determine the outermost location to copy each array group
  - Each group should be copied at most twice
  - Before entering the outermost loop that carries array reuse cycle
- Adjust location to ensure profitability
  - Enforce size limit on the buffer
    - **constant size => scalar replacement**
  - Ensure reuse of copied elements
    - Move copy location inside loops that carry no reuse
- The transformation: insert copy instructions, replace array refs

Shifting of Copy Buffer

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

```
T1=B(1)
DO I = 1, N
  T2=B(I+1)
  A(I)=T1+T2
  T1=T2
ENDDO
```

- If the outermost loop carries reuse but no reuse cycle
  - Can shift values in the buffer to enable buffer reuse
  - Copying between buffers are cheaper than copying from the original array
  - Copying between registers are much cheaper than loading from memory

Array Copy: Matrix Multiplication

```
A_buf[0:m*l] = A[0:m,0:l];
for (j=0; j<n; ++j)  {
   C_buf[0:m] = C[0:m, j*m];
   for (k=0; k<l; ++k) {
      B_buf = B[k+j*l];
      for (i=0; i<m; ++i)
          C_buf[i] = C_buf[i] + alpha * A_buf[i+k*m]*B_buf;
   }
   C[0:m,j*m]=C_buf[0:m];
}
```

- Dimensionality of buffer enforced by command-line options
- Can be applied to arbitrarily shaped loop structures
- Can be applied independently or after blocking

Array Copy: Putting it together

- Array copy and scalar replacement
    - Leave out references that cannot be copied
    - Prune dependence graph, remove dep cycle
    - Apply typed fusion to group name partitions
    - Select a set of name partitions using register pressure moderation
    - For each selected partition
        - Determine the outermost position to copy
        - Determine buffer size. Use scalars if possible.
        - Insert copy operations
        - Replace array references with buffer references

Loop Unroll-and-Jam

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

- Can we put B(J) into a register and reuse the reference?

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

- Unroll outer loop twice and then fuse the copies of the inner loop
- Now can reuse register for B(J)
- But require one more register for A

- Goal: explore register reuse by outer loops
  - Compare to loop blocking
    - Different iterations of outer loop unrolled
  - Often called register blocking
    - May increase register pressure at the innermost loop
- Transformation: two alternative ways to get the combined result
  - Unroll an outer loop, apply multi-level loop fusion to the unrolled loops
  - Strip-mine outer loop, interchange strip loop inside, then unroll strip loop

Safety of Unroll-and-Jam

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

- Apply unroll-and-jam

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

- This is wrong!

- Direction vector: (<,>)
  - This makes loop interchange illegal
- Unroll-and-Jam is similar to blocking
  - It must be safe to interchange the strip traversing outer loop with the inner loop

Unroll-and-Jam + Scalar Repl

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

```
DO I = 1, N*2, 2
    s0 = A(I)
    s1 = A(I+1)
    DO J = 1, M
        t = B(J)
        s0 = s0 + t
        s1 = s1 + t
    ENDDO
    A(I) = s0
    A(I+1) = s1
ENDDO
```

- Reduce the number of memory loads by half

Unroll-and-jam Example

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

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

Loop Interchange

- The order of a loop nest affect the effectiveness of register optimization

Original:

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

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

Optimized:

```
DO I = 2,N
   DO J = 1, M
      R1 = A(J,I-1)
      R2 = A(J,I)
      R2 = R2+R1
      A(J,I)=R2
   ENDDO
ENDDO
```

```
DO J = 1,M
   R1 = A(J,1)
   DO I = 2,N
      R2=A(J,I)
      R2=R2+R1
      A(J,I)=R2
      R1=R2
   ENDDO
ENDDO
```

- Want loops that carry dependence at innermost position

Loop Fusion

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

DO I = 1,N
   B(I) = C(I) - D(I)
ENDDO
```

If we fuse these loops, we can reuse operations in registers:

```
DO I = 1,N
   A(I) = C(I) + D(I)
   B(I) = C(I) - D(I)
ENDDO
```