Loop Invariant Code Motion

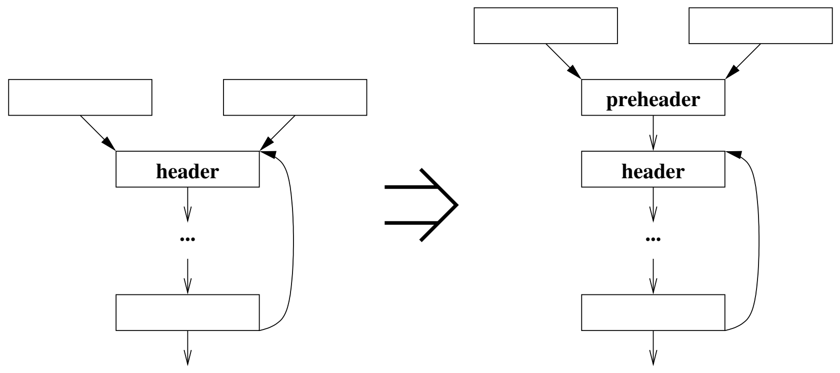
- Loop invariant code motion moves loop invariant computations out of a loop.
- VPO processes loops in the order of innermost first.
  - Loop invariant code motion requires the allocation of registers.
  - An invariant item for a loop nest will be moved out of all loops using this strategy.
- This optimization requires a preheader for the loop.

Preheader

- A preheader is a basic block that is a predecessor to the header of the loop and is not in the loop.
- If the header of a loop has more than one such predecessor, then a new (unique) preheader block is created (if an optimization needs it).
- A preheader can also only have a single successor, which is the loop header.

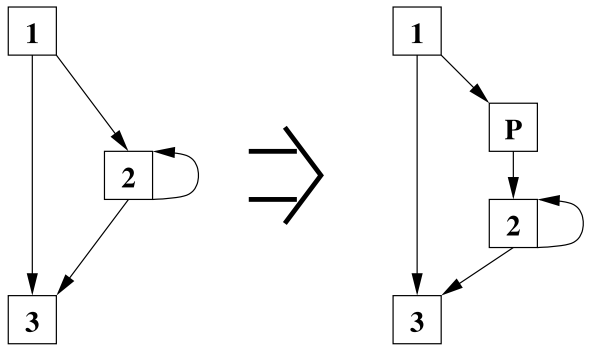
Preheader (cont.)

- Why should there be only a single preheader to a loop? It provides a single point to insert code prior to the loop.



Preheader (cont.)

- Why should the preheader have only the header as its single successor? Code moved out of the loop would only be executed when necessary.



Loop Invariant Instructions

1. An instruction is marked invariant if the entire source (every operand) is constant or is defined outside of the loop.
2. Repeat step 3 until no more instructions are marked invariant.
3. An instruction is marked invariant if source operands are constant, defined outside of the loop, or the definitions of the operands were also marked invariant.

Loop Invariant Instructions (cont.)

- Why require multiple passes?

```
for (i = 0; i < 5; i++) {
   y = 5;
   z = y+6;
}
```

- First pass determines the assignment to y is invariant.
- Second pass determines that the assignment to z is invariant.

Loop Invariant Instructions (cont.)

- The block containing the instruction must dominate all exits of the loop.

```
y = 3;
for (i = 0; i < 5; i++)
   if (...)
      y = 5; // is not loop invariant
printf("%d\n", y);
```

Loop Invariant Instructions (cont.)

- No other instruction in the loop can assign a value to the same destination.

```
for (i = 0; i < 5; i++) {
   y = 3;    // is not loop invariant
   if (...)
      y = 5;
   a = a + y;
}
printf("%d\n", a);
```

Loop Invariant Instructions (cont.)

- There is no use of the item defined by the instruction other than from the instruction itself.

```
y = 3;
for (i = 0; i < 5; i++) {
   printf("%d\n", y);
   // use of y precedes next stmt
   y = 5;
}
```

Loop Invariant References

- VPO not only attempts to move out loop-invariant assignments, but also loop-invariant sources (memory references or expressions).
- Hoisting loop-invariant sources requires:
  - assigning the loop invariant source to a register in the preheader
  - replacing the loop invariant source in the original instruction with this register

Induction Variables

- An induction variable is a variable in a loop where the next value is an increment or decrement by some constant (can induce the next value from the current value).
- A basic induction variable is an induction variable whose only assignments within the loop are of the form $i = i\ (+|-)\ c$, where c is a constant.

Loop Strength Reduction

- Reduction in strength is replacing a more expensive operation with a cheaper one.

  `x*2 => x<<1`
- Loop strength reduction allows multiply and/or left shift operations on induction variables to be replaced with additions.
  - Left shift operations are commonly used to index into arrays.
  - Multiplies are used to index into arrays of structs, where the size of each element is not a power of two.

Example of Loop Strength Reduction

- Below is an example of loop strength reduction performed at the source code level.
- Note the left shift operation is now avoided.

```
   for (i = 0; i < 100; i++)
      a[i] = 0;
=>
   for (p = &a, i = 0; i < 100; p++, i++)
     *p = 0;
```

Basic Induction Variable Elimination

- After applying loop strength reduction, we often find that the only use of some basic induction variables are in comparisons for conditional branches.
- We can often replace the comparison of that induction variable with a comparison of another.
- After making the replacement, we can often delete any increments to the original basic induction variable.

Example of Basic Induction Variable Elimination

- By changing the exit condition, we can eliminate the induction variable.

```
   for (p = &a, i = 0; i < 100; p++, i++)
      *p = 0;
=>
   for (p = &a, i = 0; p < &a+400; p++, i++)
      *p = 0;
=>
   for (p = &a; p < &a+400; p++)
      *p = 0;
```

Preconditions for Loop Strength Reduction

- VPO requires that loop-invariant code motion be performed before loop strength reduction. This analysis will result in determining which registers are loop invariant.
- VPO also requires that live variable information be calculated before loop strength reduction, which requires the allocation of new registers.

Information Associated with Each Induction Variable

- Associated with each induction variable is a triple (i, c, d), which is of the form $(c * i + d)$.
- VPO refers to these fields as:
  - family: a basic induction variable from which the current induction variable is associated
  - cee: must be an integer constant since it is used to determine how much to increment
  - dee: string of loop invariant values

Instruction Scheduling

- Reorders instructions in an attempt to:
  - minimize pipeline stalls
  - issue operations in parallel in a multiple issue architecture

Instruction Scheduling within a Basic Block to Minimize Stalls Algorithm

- Build a DAG.
  - Each node is an instruction.
  - Each edge represents a dependence (true or false) between two instructions.
  - Each edge has a weight that represents the cycles before the dependent instruction can execute.

Instruction Scheduling within a Basic Block to Minimize Stalls Algorithm (cont.)

- Initialization of Scheduling Algorithm.
  - Start with an empty list of scheduled instructions.
  - Assign a weight to each node that is the maximum of the weights of the incoming edges. If no incoming edges, then the node is assigned a weight of zero.
  - The candidate set of nodes are those that have no incoming edges.

Instruction Scheduling within a Basic Block to Minimize Stalls Algorithm (cont.)

- Scheduling Algorithm.
  - Select a node from the candidate set with the lowest weight and place the instruction at the end of the list of scheduled instructions. If ties, then chose node which has the greatest cost to reach its leaves.
  - Subtract 1 from the weights of the candidate set nodes that have no dependence with the instruction just issued.
  - Remove the selected node from the DAG.
  - Iterate if any remaining instructions to be scheduled.

Advanced Instruction Scheduling

- To reduce restrictions on moving instructions, a compiler can use:
  - register renaming
  - adjustment of offsets in memory references
- Many advanced techniques schedule instructions across basic block boundaries.

Instruction Scheduling Example

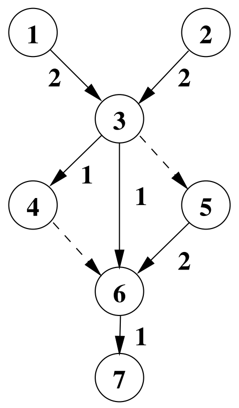
source code

```
a = b + c;
x = a - d;
```

instructions

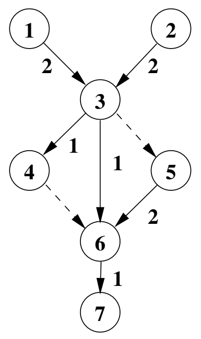
```
1. r[1]=M[b];
2. r[2]=M[c];
3. r[1]=r[1]+r[2];
4. M[a]=r[1];
5. r[2]=M[d];
6. r[1]=r[1]-r[2];
7. M[x]=r[1];
```

dependency DAG



Instruction Scheduling Example (cont.)

dependency DAG



scheduled, no renaming

```
1. r[1]=M[b];
2. r[2]=M[c];
stall
3. r[1]=r[1]+r[2];
5. r[2]=M[d];
4. M[a]=r[1];
6. r[1]=r[1]-r[2];
7. M[x]=r[1];
```

scheduled with renaming

```
1. r[1]=M[b];
2. r[2]=M[c];
5. r[3]=M[d];
3. r[1]=r[1]+r[2];
4. M[a]=r[1];
6. r[1]=r[1]-r[3];
7. M[x]=r[1];
```

Filling Delay Slots

- Filling delay slots exploits an architectural feature where a transfer of control does not take place until after the following instruction is fetched and executed.

```
Branch      IF  ID  EX  MEM WB
Delay Slot      IF  ID  EX  MEM WB
Target              IF  ID  EX  MEM WB
```

- Instructions can be used to fill the delay slot from
    - before the transfer of control,
    - the fall through block of a conditional branch, or
    - the target block of the transfer of control.

Filling Delay Slots

```
    r[2]=M[a];
    IC=r[3]?0;
    PC=IC<0,L5;
    NL=NL;
 =>
    IC=r[3]?0;
    PC=IC<0,L5;
    r[2]=M[a];
```

Other Code-Improving Transformations

- loop reordering
    - reorders iterations in a loop nest
- loop restructuring
    - changes the structure of the loop, but leaves the order of the iterations unchanged
- memory access transformations
    - changes how data or code is accessed in memory
- function call transformations
    - reduces the overhead of function calls
- exploitation of machine-dependent architectural features
    - special code-improving transformations so machine specific features can be exploited

Loop Reordering Transformations

- loop interchange
- loop reversal
- loop tiling or blocking
- loop distribution
- loop fusion

Loop Interchange

- Changes the position of two loop statements in a perfect loop nest.
- Often used to improve spatial locality.

```
    for (j = 0; j < n; j++)
       for (i = 0; i < m; i++)
          total += a[i][j];
=>
    for (i = 0; i < m; i++)
       for (j = 0; j < n; j++)
          total += a[i][j];
```

Loop Reversal

- Changes the direction in which the loop traverses its iteration range.
- Can be used to reduce the cost of the loop branch.

```
    for (i = 0; i < 100; i++)
       ...
=>
    for (i = 99; i != 0; i--)
       ...
```

Loop Reversal (cont.)

- Can also be used to improve temporal locality.

```
    for (i = 1; i < 1000; i++)
       a[i] = a[i-1]+b[i];
    for (i = 0; i < 1000; i++)
       c[i] = c[i]+a[i];
=>
    for (i = 1; i < 1000; i++)
       a[i] = a[i-1]+b[i];
    for (i = 999; i >= 0; i++)
       c[i] = c[i]+a[i];
```

Loop Tiling or Blocking

- Divides the iteration space into tiles that are the size of a level of the memory hierarchy and maximize accesses to the data in the tile before it is replaced.
- This transformation improves temporal locality.

Simple Example of Blocking

- Access n elements of b before reuse.

```
for i:=1 to n
   for j:=1 to n
      c[i,j] = a[i,j]*b[j];
```

- Now access s elements of b before reuse.

```
for jj := 1 to n by s
   for i:=1 to n
      for j:=jj to min(jj+s-1,n)
         c[i,j] = a[i,j]*b[j];
```

Loop Tiling or Blocking (cont.)

- Good reuse of a[i,j]. But a row of the "c" and "b" arrays gets reused in the next iteration of the middle and outer loop, respectively. When a large amount of data is accessed, the elements in these rows may be replaced before they can be reused.

```
for i:=1 to n
   for j:=1 to n
      for k:=1 to n
         c[i,k] += a[i,j]*b[j,k];
```

Loop Tiling or Blocking (cont.)

- Now iterations from the loops of the outer dimensions are executed before completing all the iterations of the inner loop.

```
for jj:=1 to n by s
   for kk:=1 to n by s
      for i:=1 to n
         for j:=jj to min(jj+s-1,n)
            for k:=kk to min(kk+s-1,n)
               c[i,k] += a[i,j]*b[j,k];
```

Loop Distribution

- Breaks a single loop into multiple loops.
- Each new loop has the same iteration space as the original, but contains a subset of the statements.

```
for (i = 1; i < n; i++) {
   a[i] = a[i] + c;
   b[i] = b[i] + b[i-1];
}
```

- Can execute iterations of 1st loop below in parallel.

```
for (i = 1; i < n; i++)
   a[i] = a[i] + c;
for (i = 1; i < n; i++)
   b[i] = b[i] + b[i-1];
```

Loop Fusion

- Opposite of loop distribution.
- Can be used to reduce the loop overhead.

```
   for (i = 0; i < 100; i++)
      a[i] = 0;
   for (i = 0; i < 200; i++)
      b[i] = c[i];
=>
   for (i = 0; i < 100; i++) {
      a[i] = 0;
      b[i] = c[i];
   }
   for (i = 100; i < 200; i++)
      b[i] = c[i];
```

Loop Fusion (cont.)

- Can be used to improve temporal locality.

```
   for (i = 0; i < 200; i++)
      a[i] = b[i]*c[i];
   for (i = 0; i < 100; i++)
      a[i] = a[i]+d[i];
=>
   for (i = 0; i < 100; i++) {
      a[i] = b[i]*c[i];
      a[i] = a[i]+d[i];
   }
   for (i = 100; i < 200; i++)
      a[i] = b[i]*c[i];
```

Loop Fusion (cont.)

- Now the fused loop can be improved.

```
   for (i = 0; i < 100; i++) {
      a[i] = b[i]*c[i];
      a[i] = a[i]+d[i];
   }
   for (i = 100; i < 200; i++)
      a[i] = b[i]*c[i];
=>
   for (i = 0; i < 100; i++)
      a[i] = b[i]*c[i]+d[i];
   for (i = 100; i < 200; i++)
      a[i] = b[i]*c[i];
```

Loop Restructuring Transformations

- unswitching
- loop unrolling
- software pipelining
- loop collapsing
- loop peeling
- loop splitting
- loop inversion

Unswitching

- Unswitching moves a loop-invariant conditional branch out of a loop.

```
   for (i = 0; i < 100; i++)
      if (g == 1)
         a[i] = b[i];
      else
         a[i] = c[i];
=>
   if (g == 1)
      for (i = 0; i < 100; i++)
         a[i] = b[i];
   else
      for (i = 0; i < 100; i++)
         a[i] = c[i];
```

Loop Unrolling

- Means that multiple copies of a loop body are made within the loop.
- Requires knowing the number of loop iterations statically or dynamically.
- Reduces loop overhead and provides additional scheduling opportunities by increasing the size of the header and tail blocks in the loop.
- Typically is only performed on the innermost loops of a program.

Loop Unrolling Example

```
   for (i = 0; i < n; i++)
      a[i] = b[i]+c[i];
=>
   for (i = 0; i < n%4; i++)
      a[i] = b[i]+c[i];
   for (; i < n; i += 4) {
      a[i] = b[i]+c[i];
      a[i+1] = b[i+1]+c[i+1];
      a[i+2] = b[i+2]+c[i+2];
      a[i+3] = b[i+3]+c[i+3];
   }
```

Software Pipelining

- The original iterations of a loop are rescheduled where different parts are performed in different iterations.
- Requires startup (prologue) code before the loop.
- Requires completion (epilogue) code after the loop.
- Provides opportunities for better scheduling since instructions in different iterations are often independent.

Software Pipelining Example

- original code

```
    i = 0;
L1: r1 = a[i];    -- Li
    r1 = r1 + c;  -- Ai
    a[i] = r1;    -- Si
    i++;          -- incr
    if < n iters, goto L1;
```

- pipeline diagram assuming 2 cycle load and a 5 cycle FP addition

```
Li  IF ID EX DC1   DC2   FWB
Ai     IF ID stall stall FEX FEX   FEX   FEX   FEX FWB
Si        IF stall stall ID  stall stall stall EX  DC1 DC2 WB
incr         stall stall IF  stall stall stall ID  EX  DC1 DC2 WB
```

Software Pipelining Example (cont.)

- original code after scheduling

```
    i = 0;
L1: r1 = a[i];    - Li
    r1 = r1 + c;  - Ai
    i++;          - incr
    a[i] = r1;    - Si
    if < n iters, goto L1;
```

- pipeline diagram

```
Li   IF ID EX DC1   DC2   FWB
Ai      IF ID stall stall FEX FEX FEX   FEX   FEX FWB
incr       IF stall stall ID  EX  DC1   DC2   WB
Si            stall stall IF  ID  stall stall EX  DC1 DC2 WB
```

Software Pipelining Example (cont.)

- scheduled original code with stalls depicted

```
    i = 0;
L1: r1 = a[i];    -- Li
    stall
    stall
    r1 = r1 + c;  -- Ai
    i++;
    stall
    stall
    a[i] = r1;    -- Si
    if < n iters, goto L1;
```

Software Pipelining Example (cont.)

- after software pipelining once

```
    i = 0;
    r1 = a[0];    - L0
    r1 = r1 + c;  - A0
L1: a[i] = r1;    - Si
    r1 = a[i+1];  - Li+1
    r1 = r1 + c;  - Ai+1
    i++;
    if < n-1 iters, goto L1;
    a[n-1] = r1;  - Sn-1
```

Software Pipelining Example (cont.)

- after register renaming

```
     i = 0;
     r1 = a[0];    - L0
     r1 = r1 + c; - A0
L1:  a[i] = r1;    - Si
     r2 = a[i+1]; - Li+1
     r1 = r2 + c; - Ai+1
     i++;
     if < n-1 iters, goto L1;
     a[n-1] = r1; - Sn-1
```

Software Pipelining Example (cont.)

- after scheduling instructions

```
     r1 = a[0];    - L0
     i = 0;
     r1 = r1 + c; - A0
L1:  r2 = a[i+1]; - Li+1
     a[i] = r1;    - Si
     r1 = r2 + c; - Ai+1
     i++;
     if < n-1 iters, goto L1;
     a[n-1] = r1; - Sn-1
```

Software Pipelining Example (cont.)

- after scheduling instructions with stalls depicted

```
     r1 = a[0];    -- L0
     stall
     i = 0;
     r1 = r1 + c; -- A0
     stall
     stall
L1:  r2 = a[i+1]; -- Li+1
     a[i] = r1;    -- Si
     stall
     r1 = r2 + c; -- Ai+1
     i++;
     if < n-1 iters, goto L1;
     stall
     a[n-1] = r1; -- Sn-1
```

Software Pipelining Example (cont.)

- after software pipelining again

```
     r1 = a[0];    - L0
     i = 0;
     r1 = r1 + c; - A0
     r2 = a[1];    - L1
L1:  a[i] = r1;    - Si
     r1 = r2 + c; - Ai+1
     r2 = a[i+2]; - Li+2
     i++;
     if < n-2 iters, goto L1;
     a[n-2] = r1; - Sn-2
     r1 = r2 + c; - An-1
     a[n-1] = r1; - Sn-1
```

Software Pipelining Example (cont.)

- after renaming and scheduling

```
         r1 = a[0];    - L0
         i = 0;
         r1 = r1 + c; - A0
         r2 = a[1];    - L1
     L1: a[i] = r1;    - Si
         r1 = r2 + c; - Ai+1
         r2 = a[i+2]; - Li+2
         i++;
         if < n-2 iters, goto L1;
         r2 = r2 + c; - An-1
         a[n-2] = r1; - Sn-2
         a[n-1] = r2; - Sn-1
```

Software Pipelining Example (cont.)

- with stalls depicted

```
         r1 = a[0];   -- L0
         i = 0;
         stall
         r1 = r1 + c; -- A0
         r2 = a[1];   -- L1
         stall
         stall
     L1: a[i] = r1;   -- Si
         r1 = r2 + c; -- Ai+1
         r2 = a[i+2]; -- Li+2
         i++;
         if < n-2 iters, goto L1;
         r2 = r2 + c; -- An-1
         a[n-2] = r1; -- Sn-2
         stall
         stall
         a[n-1] = r2; -- Sn-1
```

Loop Collapsing

- Combines a loop nest into a single loop.
- Can reduce loop overhead.

```
       int a[100][200];
       ...
       for (i = 0; i < 100; i++)
          for (j = 0; j < 200; j++)
             a[i][j] = 0;
    =>
       int a[20000];
       ...
       for (i = 0; i < 20000; i++)
          a[i] = 0;
```

Loop Peeling

- Separates a small number of iterations from the beginning or end of a loop.
- Can be used to eliminate branches, remove dependences, or enable loop fusion.

```
       for (i = 0; i < m; i++)
          if (i == 0)
             a[i] = b[i];
          else
             a[i] = a[i-1]+b[i];
    =>
       a[0] = b[0];
       for (i = 1; i < m; i++)
          a[i] = a[i-1]+b[i];
```

Loop Splitting

- Splits loop by iterations into separate loops.
- Has similar benefits as loop peeling.

```
   for (i = 0; i < 100; i++)
      if (i == m)
         S1;
      else
         S2;
=>
   for (i = 0; i < min(m,100); i++)
      S2;
   if (m <= 100)
      S1;
   for (i = m+1; i < 100; i++)
      S2;
```

Loop Inversion

- Places the loop exit test at the end of the loop instead of at the beginning.
- This strategy eliminates the execution of an unconditional jump in the loop.

```
   L2: IC=r[2]?100;
       PC=IC>=0,L5;
       ...
       PC=L2;
   L5:
=>
       PC=L2;
   L3: ...
   L2: IC=r[2]?100;
       PC=IC<0,L3;
   L5:
```

Memory Access Transformations

- array padding
- scalar expansion
- scalar replacement
- recurrence elimination
- packing of fields
- array layout
- fusing and splitting
- code positioning

Array Padding

- Unused data locations are inserted between arrays or within arrays.
- Can be used to reduce conflict misses in a cache or bank conflicts in cache and memory.

```
   double a[1024], b[1024];
   ...
   for (i = 0; i < 1024; i++)
      sum += a[i]*b[i];
=>
   double a[1024], pad[16], b[1024];
   ...
   for (i = 0; i < 1024; i++)
      sum += a[i]*b[i];
```

Scalar Expansion

- Expands a scalar into an array.
- This transformation is used to remove dependences between loop iterations to exploit loop-level parallelism.

```
   for (i = 0; i < 100; i++) {
      c = b[i];
      a[i] = a[i]+c;
   }
=>
   int T[100];

   for (i = 0; i < 100; i++) {
      T[i] = b[i];
      a[i] = a[i]+T[i];
   }
```

Scalar Replacement

- Replaces a loop-invariant array element with a scalar in a loop.
- Scalars can more easily be allocated to registers.

```
   for (i = 0; i < n; i++)
      for (j = 0; j < n; j++)
         total[i] = total[i]+a[i][j];
=>
   for (i = 0; i < n; i++) {
      T = total[i];
      for (j = 0; j < n; j++)
         T = T+a[i][j];
      total[i] = T;
   }
```

Recurrence Elimination

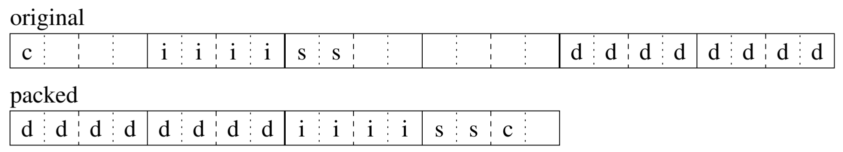
- Eliminates redundant memory accesses across loop iterations.

```
   for (i = 2; i < n; i++)
      x[i] = z[i]*(y[i] - x[i-1]);
=>
   reg = x[1];
   for (i = 2; i < n; i++)
      x[i] = reg = z[i]*(y[i] - reg);
```

Packing of Fields

- Fields within a record or a struct can be reordered to save space.
- Benefit is due to alignment requirements.

```
struct {                  struct {
   char c;                   double d;
   int i;        =>          int i;
   short s;                  short s;
   double d;                 char c;
} a[10000];               } a[10000];
```



Array Layout Transformations

- Sometimes loops cannot be interchanged to improve spatial locality.
  - Data dependences may prevent the interchange.
  - The loops may not be perfectly nested.
- The idea is to designate an array as either row-major or column-major order.
- This approach is only feasible at a high level before the code is actually generated.

Array Fusion

- This transformation is performed to improve spatial locality when two corresponding elements of different arrays are accessed close in time.
- The effect is to convert independent arrays to an array of structs.

```
   int a[1000], b[1000];

   for (i = 0; i < 1000; i++)
      ? = a[i]+b[i];
=>
   struct { int a; int b; } s[1000];

   for (i = 0; i < 1000; i++)
      ? = s[i].a + s[i].b;
```

Structure Splitting

- Opposite of array fusion.
- Converts an array of records into separate arrays.
- Useful when some fields in a struct or record are not commonly referenced at the same time as other fields in that record.

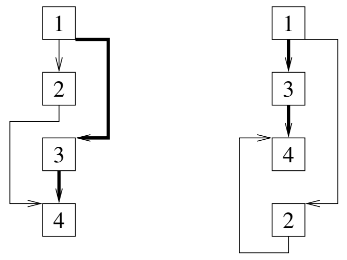
```
   struct { int a; int b; } s[1000];

   for (i = 0; i < 1000; i++)
      ? = s[i].a;
   ...
   for (i = 0; i < 1000; i++)
      ? = s[i].b;
=>
   int a[1000], b[1000];

   for (i = 0; i < 1000; i++)
      ? = a[i];
   ...
   for (i = 0; i < 1000; i++)
      ? = b[i];
```

Code Positioning

- Reorder the basic blocks in a function or program to improve spatial locality to decrease delays due to instruction fetch misses.
- Can also be used to reduce the number of unconditional jumps executed and branches taken.



Function Call Transformations

- inlining
- cloning
- tail recursion elimination
- function memoization
- leaf function optimization

Inlining

- Replaces a call with the copy of the function body.
  - advantages
    - Eliminates call and return overhead.
    - Often enables other code-improving transformations.
  - disadvantages
    - Increases code size.
    - May increase instruction cache misses.

Cloning

- Creates copies of a function, where each copy has different constant arguments.
- Enables other code improving transformations with less code growth.

```
   b = f(a, 2);
   ...
   int f(int x, int factor) {
      return x*factor;
   }
=>
   b = f_2(a);
   ...
   int f_2(int x) {
      return x<<1;
   }
```

Tail Recursion Elimination

- A function is tail recursive if it calls itself just before returning.
- The recursive call can be replaced with a jump to the top of the function.

```
   int inarray(int a[], int x, int i, int n) {
      if (i == n) return false;
      else if (a[i] == x) return true;
      else return inarray(a, x, i+1, n);
   }
=>
   int inarray(int a[], int x, int i, int n) {
   top:
      if (i == n) return false;
      else if (a[i] == x) return true;
      else { i++; goto top; }
   }
```

Function Memoization

- Traditionally uses a software cache to remember the result of a function based on its input values.
- The function must have no side effects (just calculates a return value).
- A performance benefit only occurs when the hit rate is high and the cost of recomputing the result is greater than the time to retrieve the memoized result.
- Special hardware caches have been proposed to memoize results of long-running arithmetic operations, speculatively predict values that are loaded from memory, and allow more efficient memoization of code segments.

Leaf Function Optimization

- The back end of a compiler can recognize when a function has no calls and all of its variables can be allocated to scratch (caller-save) registers.
- Can avoid allocating another register window for machines like the SPARC.
- Can avoid adjusting the stack pointer to allocate a stack frame (activation record).

Exploitation of Machine-Dependent Architectural Features

- instruction scheduling
- filling delay slots
- exploiting multiple functional units for instruction-level parallelism
- if conversion
- prefetching

Instruction Scheduling

- Avoid pipeline stalls by rearranging the order of instructions.
- Most common and simplest within a single basic block.
- There are transformations to avoid pipeline stalls within frequently executed blocks by using code duplication.
  - software pipelining
  - loop unrolling
  - superblock formation

Filling Delay Slots

- Delayed instructions do not take place until after one or more instructions following the delayed instruction execute.
- Most common types:
  - delayed branches
  - delayed loads

Transformations to Exploit Instruction-Level Parallelism

- architectures that support ILP
  - VLIW
  - superscalar (both in-order and out-of-order)
- increase distance between dependences
  - software pipelining
- increase the size of basic blocks
  - loop unrolling
  - superblocks and hyperblocks
  - if conversion

If Conversion

- Some architectures have predicated instructions, where the instruction result will only be assigned if the predicate is true.
- If conversion is the process of converting conditionally executed code into predicated code (converting control dependences into data dependences).
- This has the effect of increasing the size of basic blocks and offers new opportunities for instruction scheduling and exploiting instruction-level parallelism.
- Care must be taken to not make performance worse.

Prefetching

- Some processors have machine instructions to cause data to be prefetched into a cache line.
- The goal is to start fetching the data before it is needed in order to reduce cache miss delays when the data needs to be loaded.
- Care must be taken to avoid degrading performance by the additional execution of the prefetch instructions and potentially polluting the cache with data that is not referenced.