CSc 553

Principles of Compilation

34 : Memory Hierarchy Optimization

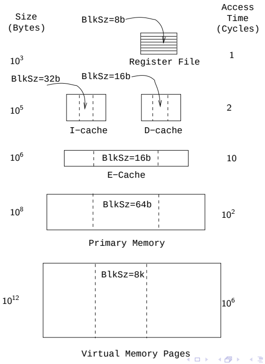
Department of Computer Science
University of Arizona

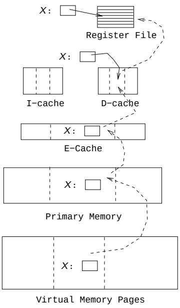
collberg@gmail.com

Memory Hierarchy

Memory Hierarchy I

- Memory is organized hierarchically. Storage at the bottom of the hierarchy is large and slow. Storage at the top of the hierarchy is small and fast.
- Accessing a memory word X could result in the following: Swap in VM page containing X $\longrightarrow$ Load memory line containing X into E-cache $\longrightarrow$ Load cache line containing X into D-cache $\longrightarrow$ Load X into register.
- Notice that when moving X up the hierarchy, we don't just move X but the entire block on which X resides.
- We should try to organize our code so that it makes efficient use of every datum moved up the hierarchy.





Memory Hierarchy IV

- We will see various compiler transformations on loops that will change the data access pattern to make efficient use of loaded data. Often, the idea is to turn a stride-n access pattern (which only uses one word from each cache line per loop iteration), into a stride-1 access.
- Loading code is no different from loading data. The I-cache is of limited size, and we should make efficient use of the instructions that are loaded. Ideally, we want loop bodies to fit neatly into the I-cache. Compiler transforms can break large loops into smaller ones, and merge small loops into larger ones.

Memory Hierarchy V

- We also want to make efficient use of virtual memory. We can sort the procedures of a program so that procedures that are likely to call each other fall on the same VM page.
- Another technique is to reduce the size of procedures by splitting them into two components: the code that is likely to execute all the time (the main-line code) and the infrequently-executed code (e.g. exception-handling code). The primary components of procedures are grouped together, and the secondary components are grouped together.

Transformations

Loop Transformations

- We'll look at transformations on FOR-loops that can affect memory hierarchy utilization. The legality of these transformations depends on the loops' data dependencies.
- Some of these transformations are also used by parallelizing compilers. In general, a loop can't be parallelized (reorganized to be run on a multiprocessor machine) if it has any data dependencies. Some transformations shown here can break such dependencies so that the loop can be parallelized.
- Some of the loop transformations do not improve performance by themselves, but reorganize the loops so that they are amenable to other optimizing loop transformations.

Loop Fission

Loop Fission I

- **Loop Fission** breaks a loop into two or more independent loops. Also known as *loop distribution*.
- The smaller loops may fit better in the I-cache, may have better D-cache utilization, or can more easily be parallelized.
- Can the loop below be broken into smaller loops?

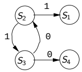
```
FOR I := 1 TO N DO
   S1:   A[I] := A[I] + B[I-1];
   S2:   B[I] := C[I-1] * X + V;
   S3:   C[I] := 1/B[I];
   S4:   D[I] := sqrt(C[I]);
ENDFOR
```

Loop Fission II

Dependencies

- $S_2\ \delta_<\ S_1$: S_2 assigns a value to B[I] that will be used by S_1 in the next iteration.
- $S_2\ \delta_=\ S_3$: S_2 assigns a value to B[I] that will be used by S_3 in the same iteration.
- $S_3\ \delta_<\ S_2$: S_3 assigns a value to C[I] that will be used by S_3 in the next iteration.
- $S_3\ \delta_=\ S_4$: S_3 assigns a value to C[I] that will be used by S_4 in the same iteration.

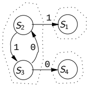
```
FOR I := 1 TO N DO
   S1: A[I]:=A[I]+B[I-1];
   S2: B[I]:=C[I-1]*X+V;
   S3: C[I]:=1/B[I];
   S4: D[I]:=sqrt(C[I]);
ENDFOR
```



Loop Fission III

- If there are no cycles in the dependency graph, we can split the loop into separate loops for each statement.
- The loops must be ordered in a topological order according to the graph.
- If the graph has cycles, the statements in each **strongly connected component** must be in the same loop.
- Two nodes n_1 and n_2 of a graph G are in the same strongly connected component C, if there is a path from n_1 to n_2 and a path from n_2 to n_1.

```
FOR I := 1 TO N DO
  S1: A[I]:=A[I]+B[I − 1];
  S2: B[I]:=C[I − 1]*X+V;
  S3: C[I]:=1/B[I];
  S4: D[I]:=sqrt(C[I]);
ENDFOR
```



Loop Fission IV

- The dependence graph has 3 strongly connected components ($[S_1]$, $[S_2, S_3]$, $[S_4]$) $\Rightarrow$ the loop can be split into 3 separate loops.
- Since the graph has edges $[S_2, S_3] \rightarrow [S_1]$ and $[S_2, S_3] \rightarrow [S_4]$, the $[S_2, S_3]$ loop has to precede the other loops.

```
FOR J := 1 TO N DO
   S2:   B[J] := C[J − 1] * X + V;
   S3:   C[J] := 1/B[J];
ENDFOR;
FOR J := 1 TO N DO
   S1:   A[J] := A[J] + B[J − 1];
ENDFOR;
FOR J := 1 TO N DO
   S4:   D[J] := sqrt(C[J]);
ENDFOR;
I := N;
```

Loop Fusion

Loop Fusion I

- Loop fusion merges two adjacent loops.
- Fusion can reduce loop overhead, increase instruction parallellism, improve locality, and improve load balance.

Original Loops

```
FOR i := 1 TO N DO
   S1:   A[i] := A[i] + k;
ENDFOR;
FOR i := 1 TO N DO
   S2:   B[i + 1] := B[i] + A[i];
ENDFOR;
```

Loops After Fusion

```
FOR i := 1 TO N DO
   S1:   A[i] := A[i] + k;
   S2:   B[i + 1] := B[i] + A[i];
ENDFOR;
```

Loop Fusion II

- The loops must have the same loop bounds.
- Two loops cannot be fused if $\exists$ a statement S_1 in the 1st loop and a statement S_2 in the 2nd loop, such that $\exists$ a dependence $S_2 \Rightarrow S_1$ in the fused loop.

```
FOR i := 1 TO N DO
   S1:   A[i] := A[i] + k;
ENDFOR;
FOR i := 1 TO N DO
   S2:   B[i+1] := B[i] + A[i+1];
ENDFOR;
          ⇓ Illegal!
FOR i := 1 TO N DO
   S1:   A[i] := A[i] + k;
   S2:   B[i+1] := B[i] + A[i+1];
ENDFOR;
```

Loop Reversal

Loop Reversal I

- **Loop reversal** runs a loop backwards.
- Reversal is legal only when there are no loop-carried dependence relations.
- Reversal can help with loop fusion. The loops below cannot be directly fused, since there would be a forward dependence between S_2 and S_3 (eg. for $i = 5$, S_3 would use the old value of C[6] rather than the new value computed by S_2.).

Original Loops

```
FOR i := 1 TO N DO
   S1:   A[i] := B[i] + 1;
   S2:   C[i] := A[i] / 2;
ENDFOR;
FOR i := 1 TO N DO
   S3:   D[i] := 1 / C[i+1];
ENDFOR;
```

Loop Reversal II

- Neither loop has any loop-carried dependencies, hence they can both be reversed. The reversed loops can be fused.

```
          ⇓ Reverse!
FOR i := N TO 1 DO
   S1:   A[i] := B[i] + 1;
   S2:   C[i] := A[i] / 2;
ENDFOR;
FOR i := N TO 1 DO
   S3:   D[i] := 1 / C[i+1];
ENDFOR;
          ⇓ Fuse!
FOR i := N TO 1 DO
   S1:   A[i] := B[i] + 1;
   S2:   C[i] := A[i] / 2;
   S3:   D[i] := 1 / C[i+1];
ENDFOR;
```

Loop Unswitching

Loop Unswitching I

- Conditional statements within a loop can reduce I-cache utilization and prevent parallelization. We can break out the if-statement and replicate the loops, to get two loops without any branches.
- If the boolean expression E is *loop invariant* then we can extract it out of the loop.

Original Loop

```
FOR i := 2 TO N DO
   S1:   A[i] := A[i] + k;
         IF E THEN
   S2:      B[i] := A[i] + C[i];
         ELSE
   S3:      B[i] := A[i − 1] + B[i − 1];
         ENDIF;
ENDFOR;
```

Loop Unswitching II

- If E could possibly throw an exception then we must guard it with a test in case the loop is never executed.

Unswitched Loop

```
IF N > 1 THEN
   IF E THEN
      FOR i := 2 TO N DO
         S1:   A[i] := A[i] + k;
         S2:   B[i] := A[i] + C[i];
      ENDFOR;
   ELSE
      FOR i := 2 TO N DO
         S1:   A[i] := A[i] + k;
         S3:   B[i] := A[i − 1] + B[i − 1];
      ENDFOR;
   ENDIF;
ENDIF;
```

Loop Peeling

Loop Peeling I

- To *peel* a loop we unroll the first (or last) few iterations.
- Peeling can remove dependencies created by the first (or last) few iterations of a loop. It can also help with loop fusion by matching the loop bounds of adjacent loops.
- The first loop below can not be parallelized since there is a flow dependence between iteration $i = 2$ and iterations $i = 3, \cdots n$.

Original Loops

```
FOR i := 2 TO N DO
   S1:   B[i] := B[i] + B[2];
ENDFOR;
FOR i := 3 TO N DO
   S2:   A[i] := A[i] + k;
ENDFOR;
```

$\Downarrow$ Peel!

```
IF N >= 2 THEN
   B[2] := B[2] + B[2];
ENDIF;
FOR i := 3 TO N DO
   S1:   B[i] := B[i] + B[2];
ENDFOR;
FOR i := 3 TO N DO
   S2:   A[i] := A[i] + k;
ENDFOR;
```

$\Downarrow$ Fuse!

```
IF N >= 2 THEN
   B[2] := B[2] + B[2];
ENDIF;
FOR i := 3 TO N DO
   S1:   B[i] := B[i] + B[2];
   S2:   A[i] := A[i] + k;
ENDFOR;
```

Loop Normalization

Loop Normalization I

- Normalization converts all loops so that the induction variable is initially 1 (or 0), and is incremented by 1 on each iteration.
- Normalization can help other transformations, such as loop fusion and peeling.

Original Loops

```
FOR i := 1 TO N DO
   S1:   A[i] := A[i] + k;
ENDFOR;

FOR i := 2 TO N+1 DO
   S2:   B[i] := A[i-1] + B[i];
ENDFOR;
```

⇓ Normalize!

```
FOR i := 1 TO N DO
   S1:   A[i] := A[i] + k;
ENDFOR;

FOR i := 1 TO N DO
   S2:   B[i+1] := A[i] + B[i+1];
ENDFOR;
```

⇓ Fuse!

```
FOR i := 1 TO N DO
   S1:   A[i] := A[i] + k;
   S2:   B[i+1] := A[i] + B[i+1];
ENDFOR;
```

Loop Interchange

Loop Interchange I

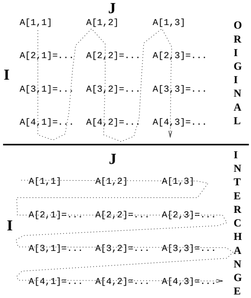
- Loop interchange moves an inner loop outwards in a loop nest. It can improve locality (and hence cache performance) by turning a stride-n access pattern into stride-1:

Original Loop

```
FOR i := 1 TO N DO
   FOR j := 1 TO N DO
      B[i] := B[i] + A[j,i];
   ENDFOR;
ENDFOR;
```

Interchanged Loop

```
FOR j := 1 TO N DO
   FOR i := 1 TO N DO
      B[i] := B[i] + A[j,i];
   ENDFOR;
ENDFOR;
```



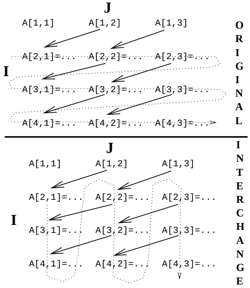
Loop Interchange III

- A loop nest of two loops can be interchanged only if there does not exist a loop dependence vector of the form $(<,>)$.
- The loops in the loop nest below can't be interchanged. The next slide shows the order in which the array elements are assigned (dashed arrows); first in the original nest and then in the interchanged nest. Solid arrows show dependencies.

This Loop Nest Can't be Interchanged

```
FOR i := 2 TO N DO
   FOR j := 1 TO N-1 DO
      A[i,j] := A[i-1,i+1];
   ENDFOR;
ENDFOR;
```

- In the interchanged loop A[2,3] is needed to compute A[3,2]. At that time A[2,3] has not been computed.



Loop Blocking

Loop Blocking I

- Also known as *loop tiling*.
- The loop below assigns the transpose of B to A. Access to A is *stride-1*, access to B is stride-n. This makes for poor locality, and the loops will perform poorly on cached machines (unless the arrays fit in the cache).
- Loop blocking improves locality by iterating over a sub-rectangle of the iteration space.
- A pair of adjacent loops can be blocked if they can legally be interchanged.

```
FOR i := 1 TO 8 DO
   FOR j := 1 TO 8 DO
      A[i,j] := B[j,i];
   ENDFOR;
ENDFOR;
```

Loop Blocking II

- To block a loop `FOR i = lo TO hi DO` we select the following constants:
  - `ts` The block size.
  - `to` The block offset ($0 \leq$ to $<$ ts). Each block will start at an iteration such that $i \bmod$ ts $=$ to.

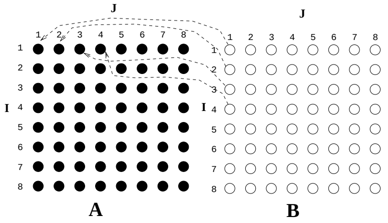
Blocked Loop

```
FOR Ti := ⌊(lo-to)/ts⌋*ts+to
      TO ⌊(hi-to)/ts⌋*ts+to BY ts DO
   FOR i := max(Ti,lo) TO min(Ti+ts-1,hi) DO
```

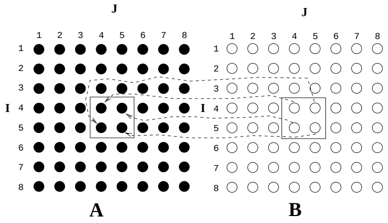
Loop Blocking III

```
FOR i := 1 TO 8 DO
   FOR j := 1 TO 8 DO
      A[i,j] := B[j,i];
   ENDFOR;
ENDFOR;
         ⇓ Block!
FOR Ti := 1 TO 8 BY 2 DO
   FOR Tj := 1 TO 8 BY 2 DO
      FOR i := Ti TO min(Ti+1, 8) DO
         FOR j := Tj TO min(Tj+1, 8) DO
            A[i,j] := B[j,i];
         ENDFOR;
      ENDFOR;
   ENDFOR;
ENDFOR;
```

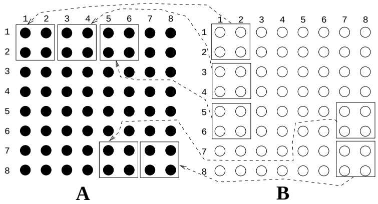
Loop Blocking IV (A) – Original Loop



Loop Blocking IV (B) – Blocked Loop



Loop Blocking IV (B) – Block Movements

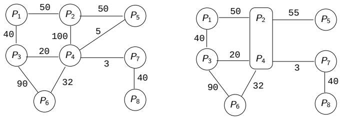


Procedure Sorting

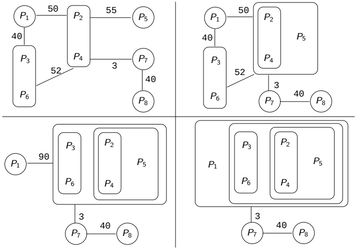
Procedure Sorting I

- The simplest way to increase VM performance is to sort the procedures of a program so that routines that are likely to call each other will fall on the same VM page.
- At link-time (or after link-time), build an un-directed call graph. Label each edge $P \rightarrow Q$ with the frequency of calls between P and Q.
- Collapse the graph in stages. At each stage select the edge $P \xrightarrow{k} Q$ with max weight k, merge nodes P and Q, collapse edges into P and Q into a single edge (adding the edge weights).
- Nodes that are merged are put on the same page.

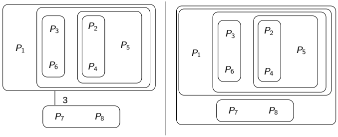
Procedure Sorting – Example (a)



Procedure Sorting – Example (b)



Procedure Sorting – Example (c)



- The final, single, node contains: $[[P_1, [P_3, P_6], [P_5, [P_2, P_4]], [P_7, P_8]]$.
- We arrange the procedures in the order $P_1, P_3, P_6, P_5, P_2, P_4, P_7, P_8$.

Homework

Exam Problem I (415.730/97)

- Consider the following loop:

```
FOR i := 1 TO n DO
    S1:   B[i] := C[i − 1] * 2;
    S2:   A[i] := A[i] + B[i − 1];
    S3:   D[i] := C[i] * 3;
    S4:   C[i] := B[i − 1] + 5;
ENDFOR
```

1. List the data dependencies for the loop. For each dependence indicate whether it is a **flow**- ($\longrightarrow$), **anti**- ($\not\longrightarrow$), or **output**-dependence ($\multimap$), and whether it is a **loop**-carried dependence or not.
2. Apply loop fission to the loop. Show the resulting loops after the transformation.

Summary

References

- David Bacon, Susan Graham, Oliver Sharp, *Compiler Transformations for High-Performance Computing*, Computing Surveys, No. 4, pp. 345–420, Dec, 1994.[1]
- Steven Muchnick, *Advanced Compiler Design & Implementation*, Chapter 20, pp. 669–704.
- Hennessy, Patterson, *Computer Architecture – A Quantitative Approach*, Section 1.7.

[1] Much of the material in this lecture has been shamelessly stolen from this article.

Summary

- Compilers use a number of loop transformation techniques to convert loops to parallelizable form.
- The same transformations can also be used to improve memory hierarchy utilization of scientific (numerical) codes.
- Nested loops can be interchanged, two adjacent loops can be joined into one (*loop fusion*), a single loop can be split into several loops (*loop fission*), etc.