# Lecture Notes on
# Loop Transformations for Cache Optimization

15-411: Compiler Design
André Platzer

Lecture 24
November 16, 2010

## 1 Introduction

In this lecture we consider loop transformations that can be used for cache optimization. The transformations can improve cache locality of the loop traversal or enable other optimizations that have been impossible before due to bad data dependencies. The same loop transformations are needed for loop parallelization and vectorization.

## 2 Loop Permutation

Loop permutation swaps the order of loops. The purpose is to pull the loops out that actually carry the data dependencies, because the inner loops will then be parallelizable. Another purpose is to swap such that the inner loops traverse arrays according to the actual memory layout of the array to reduce cache misses. That can also help with optimizations to put array contents that become constant during a loop into registers.

We assume Fortran-style column major order where B[i1,i2] is adjacent to B[i1+1,i2] in memory. Consider

```
for (i1=0; i1<n1; i1++)
  for (i2=0; i2<n2; i2++)
    A[i1] = A[i1] + B[i1,i2]
```

Here A[i1] can constantly remain in a register for n2 iterations of the inner loop. From those $n1*n2$ memory accesses to A[i1], we expect only n1 cache misses. For B[i1,i2], however, the iteration order is non-local compared to the memory layout, hence we expect $n1 * n2$ cache misses for large data.

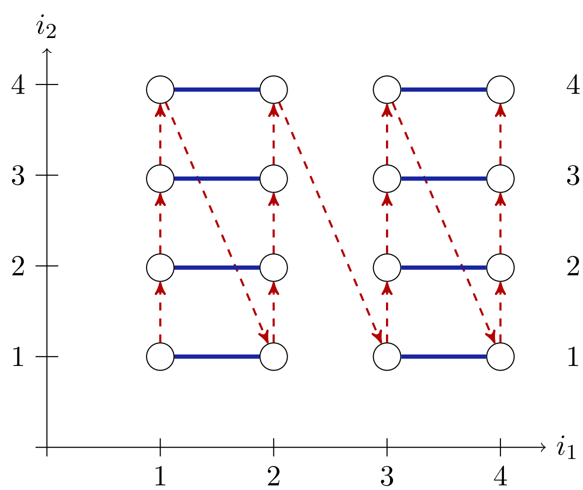
In contrast, when we swap loops

```
for (i2=0; i2<n2; i2++)     // loops swapped
  for (i1=0; i1<n1; i1++)
    A[i1] = A[i1] + B[i1,i2]
```

For A[i1], we expect $n1 * n2 * \frac{s}{cache\ size}$ cache misses, where $s$ is the data size of the array elements, because we access $n1 * n2$ times non-locally. For B[i1,i2], we similarly expect only $n1 * n2 * \frac{s}{cache\ size}$ cache misses.

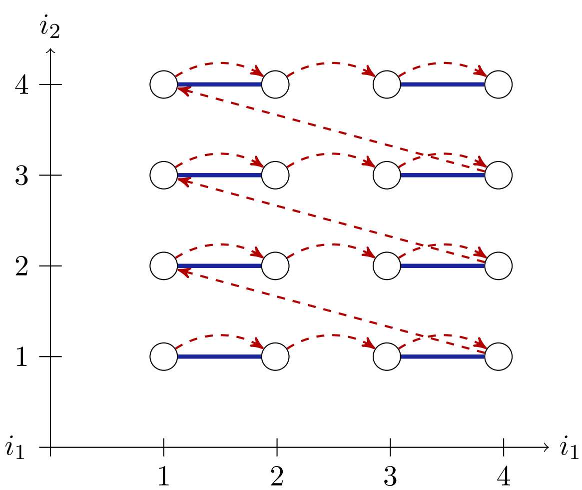
Loop permutation can have quite a remarkable effect.

```
for (i1=1; i1<=4; i1++)
  for (i2=1; i2<=4; i2++)
    A[i1,i2] = A[i1,i2] + 5
```




This loop iterates against the memory layout, which will lead to excessive cache misses on larger data.

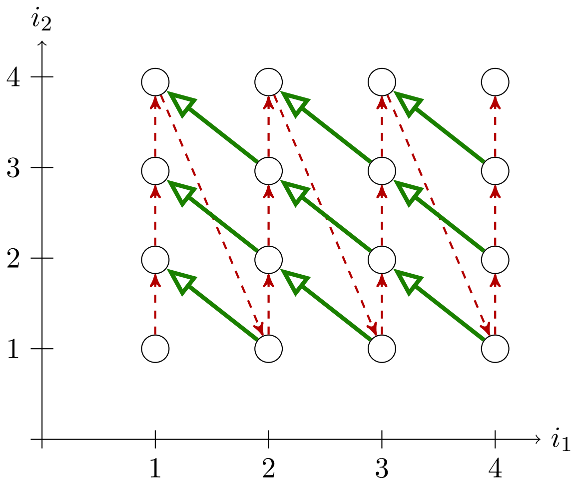
```
for (i2=1; i2<=4; i2++) // swap
  for (i1=1; i1<=4; i1++)
    A[i1,i2] = A[i1,i2] + 5
```




This loop iterates with the memory layout and uses the full cache line immediately.

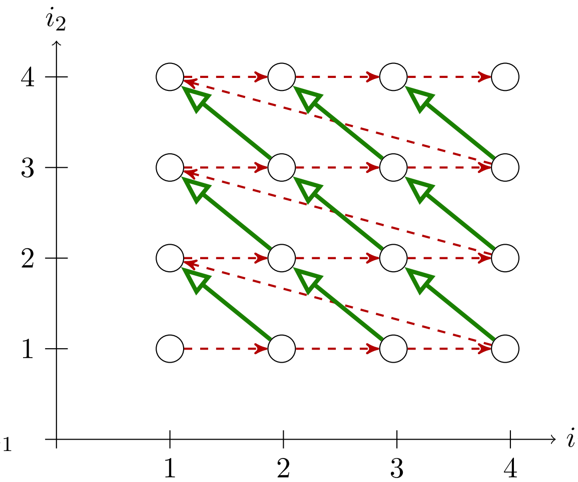
Loop permutation doesn't always have to be admissible, for instance, if we violate data dependencies by swapping loops. The dependency distance vector is permuted just like the iteration vector is. The important side condition is that the first non-zero sign of the dependency distance vector is not allowed to change.

```
for (i1=1; i1<=4; i1++)
  for (i2=1; i2<=4; i2++)
    A[i1,i2] = A[i1-1,i2+1] + 7
```




Dependency distance d=(1,-1)

```
for (i2=1; i2<=4; i2++)   // swap
  for (i1=1; i1<=4; i1++)
    A[i1,i2] = A[i1-1,i2+1] + 7
```




Dependency distance d=(-1,1)
Has different signs, thus loop swap illegal.

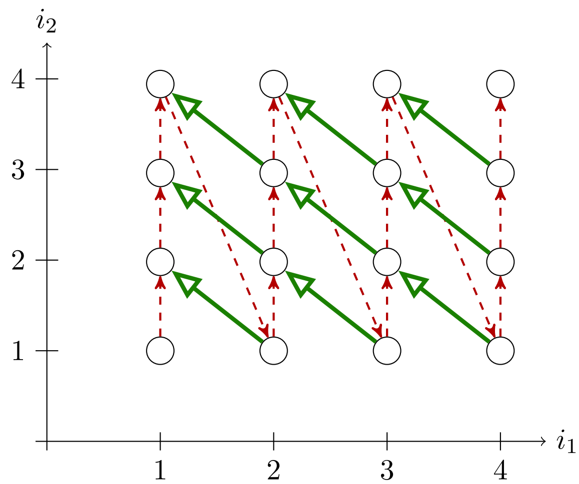
## 3 Loop Reversal

The point of loop reversal is to change the order in which a loop iterates. The primary advantage of that is that data dependencies change, which may enable other optimizations. Another advantage can be that the special ”jump if zero” (JMPZ) machine instructions can be used when loops iterate down to 0.

We can do loop reversal for loop $q$ if all data dependencies are carried by outer loops, i.e., for all dependency distances $d$ there is a $k < q$ such that $d_k \neq 0$ or $d_q = 0$. Otherwise, the direction of the dependency will be reversed by loop reversal.

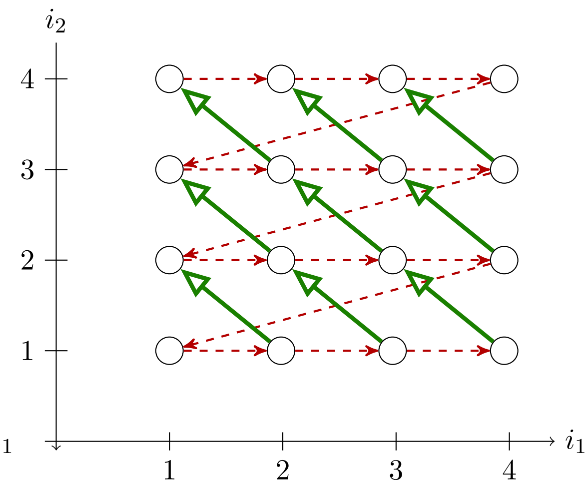
Take a look at the last loop permutation example again.

```
for (i1=1; i1<=4; i1++)
  for (i2=1; i2<=4; i2++)
    A[i1,i2] = A[i1-1,i2+1] + 7
```




Dependency distance d=(1,-1)

```
for (i1=1; i1<=4; i1++)
  for (i2=4; i2>=1; i2--) // rev.
    A[i1,i2] = A[i1-1,i2+1] + 7
```



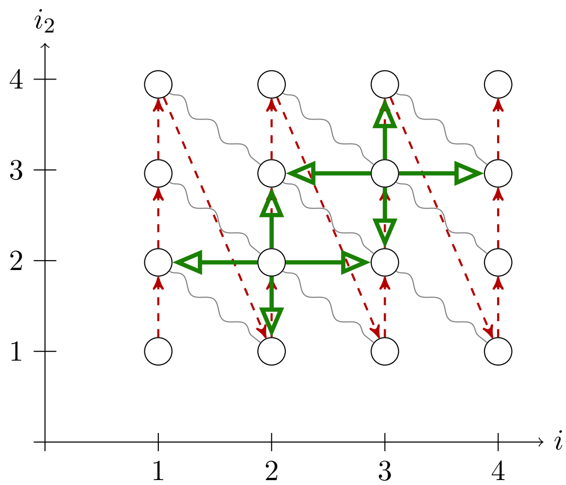

Dependency distance d=(1,1)

Note that the $i_1$ loop cannot be reversed, because that would lead to the bad dependency distance d=(-1,-1).

## 4 Loop Skewing

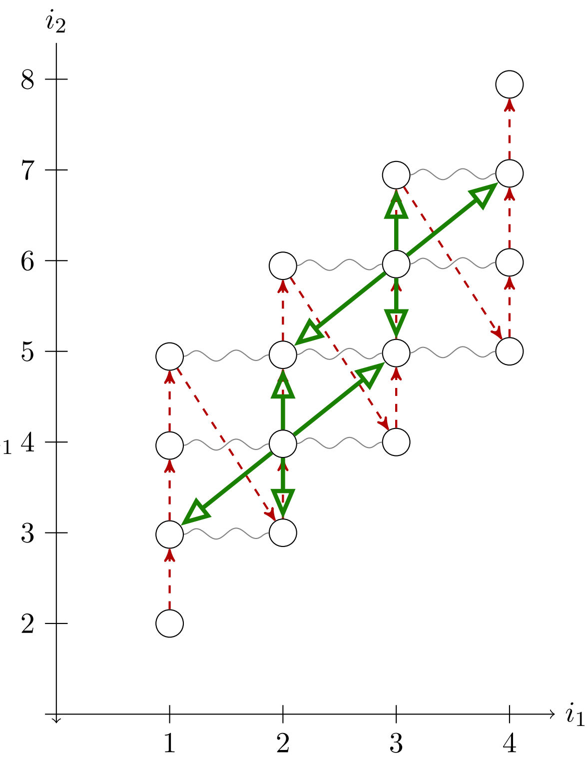
Loop skewing by a factor $f$ adds $f * i_1$ to the upper and lower bounds of inner loop $i_2$ and makes up for that by subtracting $f * i_1$ again from all array accesses of $i_2$. The advantage of doing so is that the data dependency direction changes from $d$ to $(d_1, f * (d_1 + d_2))$. By a smart choice of $f$, this change of the data dependency can enable other loop optimizations.

```
for (i1=1; i1<=4; i1++)
  for (i2=1; i2<=4; i2++)
    A[i1,i2] =
      (A[i1-1,i2]+A[i1+1,i2]
      +A[i1,i2-1]+A[i1,i2+1])/4
```




Dependency distances d=(1,0), d=(0,1).
Other dependencies d=(-1,0), d=(0,-1).
No loop can parallelize.
We cannot swap to fix this.

```
for (i1=1; i1<=4; i1++)
  for (i2=1+i1; i2<=4+i1; i2++)//skew
    A[i1,i2-i1] =
      (A[i1-1,i2-i1]+A[i1+1,i2-i1]
      +A[i1,i2-i1-1]+A[i1,i2-i1+1])/4
```




Loop skew by factor f=1
Dependency distances d=(1,1), d=(0,1).
Other dependencies d=(-1,-1), d=(0,-1).
Swapping is admissible and leads to
Dependency distances d=(1,1), d=(1,0).
Thus inner loop parallelizable.

## References