Lecture Notes on
Cache Iteration & Data Dependencies

15-411: Compiler Design
André Platzer

Lecture 23

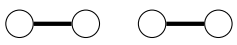
1 Introduction

Cache optimization can have a huge impact on program execution speed. It can accelerate by a factor 2 to 5 for numerical programs. Loops are the parts of the program that are generally executed most often. That is why cache optimization usually focuses exclusively on handling loops. Especially for loops that execute very often, optimizing small chunks of source code can have a fairly significant effect. Furthermore, loops often use mathematically regular access to arrays which is amenable to mathematical analysis. Cache optimization techniques are also important for vectorization and parallelization optimizations.

Some other information on cache optimization can be found in [App98, Ch 21].

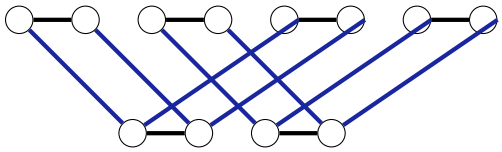
2 The Importance of Cache Optimization

For illustration purposes, take a look at a computer with a small cache of two cache lines with two data entries per cache line.



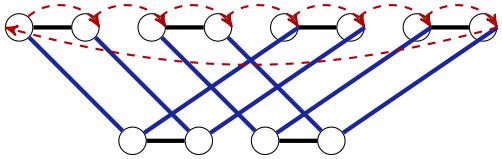
We further assume a directly mapped cache (without associativity) to simplify the presentation. We assume that a (small) array A with 8 elements has the following memory layout and maps as indicated by the solid blue lines to the cache lines. We illustrate the cache to array field association in

blue:



Cache Capacity Miss Consider the following loop that repeats the same data access in a one-dimensional array 8 times:

```
int A[8];
for(i = 0; i<8; i++)
  for(j = 0; j<8; j++)
    A[j] = ...
```

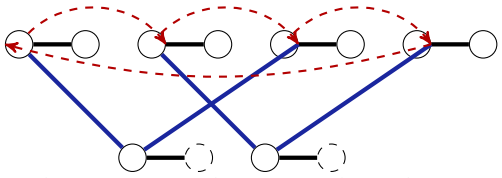


As illustrated by the dashed red iteration order, even though every array cell is used repeatedly, the data does not fit into the cache and will be reloaded twice during each repetition of the outer loop.

These 100% cache misses are caused by insufficient *cache capacity miss*. Hence, this loop should be cache-optimized to avoid suboptimal loop traversal in the cache.

Cache Line Capacity Miss Next, consider the following program

```
int A[8];
for(i = 0; i<8; i++)
  for(j = 0; j<8; j+=2)
    A[j] = ...
```

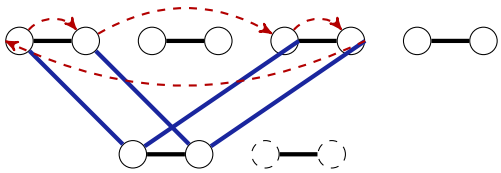


As illustrated by the dashed red iteration order, even though every array cell is used repeatedly, and even though the data would fit into the cache, unnecessary data wastes cache line space that is never used. Thus, the data will still be reloaded twice during each repetition of the outer loop.

These 100% cache misses are caused by insufficient *cache line capacity miss*. Hence, this loop should be cache-optimized to avoid suboptimal loop traversal in the cache. Here, possible cache optimizations for this particular loop traversal order also include reordering matrix elements, which is sufficient here. But that does not work if the program accesses the data in different ways in other parts of the program unless the compiler copies the matrix before. Thus, loop traversal optimization usually makes more sense.

Cache Conflict Miss Next, consider the following program

```
int A[2][4];
for(i = 0; i<8; i++)
  for(j = 0; j<1; j++)
    for(k = 0; k<1; k++)
      A[j,k] = ...
```



As illustrated by the dashed red iteration order, even though every array cell is used repeatedly, and even though the data would fit into the cache, the same cache lines are always used for all accessed data. Thus, the data will still be reloaded twice during each repetition of the outer loop.

These 100% cache misses are caused by *cache conflict miss*. Hence, this loop should be cache-optimized to avoid suboptimal loop traversal in the cache.

3 Data Dependencies

We use the following abbreviations for data dependencies between two locations ℓ and ℓ' in a program:

- $\ell \delta^t \ell'$ true data dependency (read after write)
- $\ell \delta^o \ell'$ output dependency (write after write)
- $\ell \delta^a \ell'$ anti-dependency (write after read)
- $\ell \delta^i \ell'$ input dependency (read after read)

These data dependencies can come in two flavors. Either just within a single iteration of the loop or they can be loop-carried, i.e., data dependencies between different loop iterations.

```
int A[8];
for(i = 0; i<8; i++) {
  ℓ1: A[i] = ...;
  ℓ2: x = a[i];          // ℓ1δ^t ℓ2 loop-independant
  ℓ3: y = a[i-1];        // ℓ1δ^t ℓ3 loop-carried
}
```

4 Loop Iteration Vectors

In the following we simplify the presentation. We refer to previous lectures for guarding against array access out of bounds and mostly ignore this here. We generally assume that we have perfectly nested loops (outer loops have no other statements than just the induction variable increment and the inner loop).

```
for (i1 = o1; i1 < n1; i1++)
  for (i2 = o2; i2 < n2; i2++)
    for (i3 = o3; i3 < n3; i3++)
      ...
        for (id = od; id < nd; id++) {
          loop body;
        }
```

We assume that we have already performed induction variable analysis and found basic induction variables corresponding to the respective loop nesting. We denote the above loop iteration by a single iteration vector

$$i = \begin{pmatrix} i1 \\ i2 \\ i3 \\ \vdots \\ id \end{pmatrix}$$

Loops determine an iteration order $\prec$ on the index set $\mathbb{Z}^d$. We generally restrict attention to *affine array references*, i.e., those where the index expression is an affine linear function of the iteration vector i. That is all array accesses are of the form $A[Mi + c]$ for a matrix M and vector c. For instance,

$A[3{*}i_1{+}i_2{-}1, 4{*}i_2{+}5, 2{*}i_3]$ corresponds to access of A at $\begin{pmatrix} 3 & 1 & 0 \\ 0 & 4 & 0 \\ 0 & 0 & 2 \end{pmatrix} i + \begin{pmatrix} -1 \\ 5 \\ 0 \end{pmatrix}$

Two affine array accesses $A[Mi + c]$ and $A[M'i + c']$ are called *uniform* iff $M = M'$. If there is a dependency $L\delta^t L'$ between two statements writing to uniform array access $A[Mi + c]$ and reading from uniform array access $A[Mi + c']$ with $Mi + c \prec Mi + c'$ then the uniform distance $d := c - c'$ is called *dependency distance*. This distance only makes sense in the case of uniform access, because the difference is not a constant vector otherwise. For example,

```
for (i1 = o1; i1 < n1; i1++)
  for (i2 = o2; i2 < n2; i2++)
    for (i3 = o3; i3 < n3; i3++)
      ...
        for (in = on; in < nn; in++) {
          A[M i + c] = A[M i + c'] + 5
        }
```

has dependency distance $d := c - c'$.

If the dependency distance is of the form

$$d = \begin{pmatrix} 0 \\ \vdots \\ 0 \\ d_k \\ \vdots \\ d_n \end{pmatrix} \quad \text{for some } d_k \geq 0$$

then the loop i_k carries that data dependency.

Loop i_j can be parallelized if the dependency distances d of all data dependencies satisfy

$$d_j = 0 \text{ or } d_k > 0 \text{ for some } k < j$$

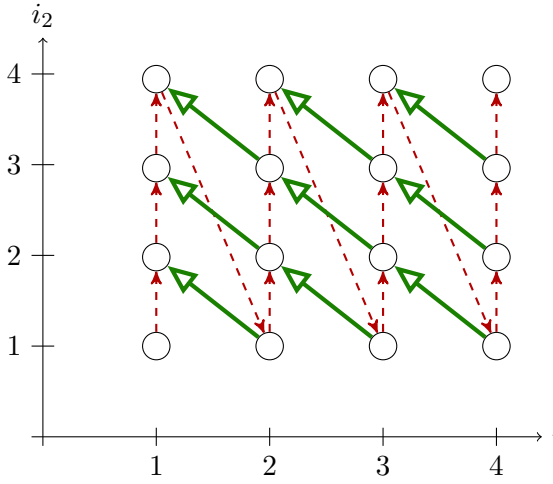
Then such a dependency either has no data dependency ($d_j = 0$) or depends on data from a past iteration of an outer loop k, and thus, from an iteration of the outer loop that will already have completed.

If these assumptions are met, we can parallelize loop i_j, for instance using SSE3 vector processing instructions. See Chapter 4 of http://www.intel.com/Assets/PDF/manual/248966.pdf. SSE3 works on various subdivisions of 128 bit data. Automatic vectorization using single instruction multiple data (SIMD) is one very important reason why Intel compilers often achieve better performance compared to gcc.

5 Data Dependencies and Loop Optimizations

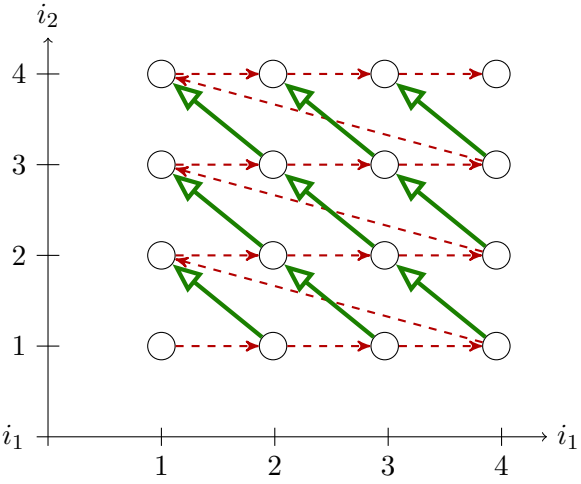
Data dependencies need to be respected in loop optimizations. The following loop uses data from the past iteration:

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



Dependency distance vector d=(1,-1)

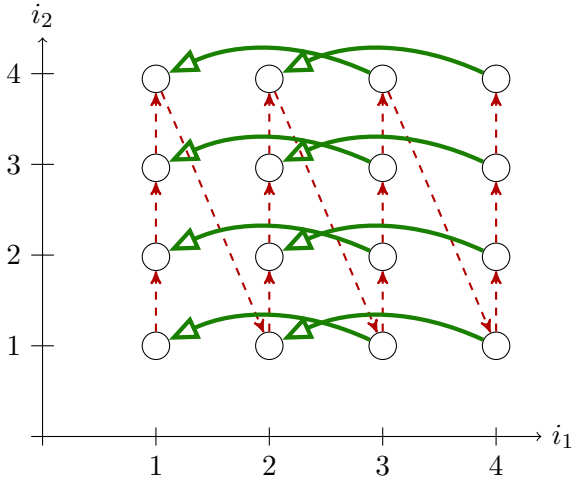
```
for (i2=1; i2<=4; i2++) // swap
  for (i1=1; i1<=4; i1++)
    A[i1,i2] = A[i1-1,i2+1]+5
```



Dependency distance vector d=(-1,1)
When swapping the loops, however, we violate the data dependency, which now depends on data from a future iteration.

Data dependencies also limit parallelization and vectorization

```
for (i1=1; i1<=4; i1++)
  for (i2=1; i2<=4; i2++)
    A[i1,i2] = A[i1-2,i2]-5
```



Dependency distance vector d=(2,0)

Because of the above data dependencies, the i_2 loop can be parallelized, but the i_1 loop cannot be parallelized.

6 Data Dependency Analysis

What we need is a checkable criterion which we can use to decide if two array references lead to a dependency or not. In practice, we restrict our attention to affine array references in perfectly nested loops. That is, given two array accesses $A[Mi + c]$ and $A[M'i + c']$ we need to find instances $i, i' \in \mathbb{Z}^d$ of the iteration vector with $i \prec i'$ such that $Mi + c = M'i' + c'$. If such instances i, i' exist that have integer values and are within the loop bounds, then the two references have a data dependency. Unfortunately, this problem is that of solving linear Diophantine equation systems, which is NP-complete.

In order to find a reasonable approximation of the dependency analysis, we first pretend all references were dependent (which is a conservative overapproximation) and then remove some dependencies if we can show that they are independent. A simple (approximate) independency check is the gcd test. Suppose we have the loop

```
for (i=0; i<10; i++)
    A[m*i+c] = A[m'*i+c']+7
```

The question is, whether there are two positions of the iteration vector (here just $\check{i}, \hat{i} \in \mathbb{Z}^1$ for the write iteration $\check{i}$ and the read iteration $\hat{i}$) such that the array accesses interfere because $m * \check{i} + c = m' * \hat{i} + c'$.

$$\begin{aligned} & m * \check{i} + c = m' * \hat{i} + c' \\ \Leftrightarrow\ & m * \check{i} - m' * \hat{i} = c' - c \end{aligned}$$

The last linear Diophantine equation can only have an integer solution if $\gcd(m, m')$ divides $c' - c$. The reason for this is that, for integers $\check{i}, \hat{i}$, the term $m * \check{i} - m' * \hat{i}$ can only take on exactly the values that are multiples of the $\gcd(m, m')$. (More formally, this is a simple property of principal ideals in Euclidean domains.)

For instance, consider

```
for (i=0; i<10; i++)
    A[2*i+2] = A[2*i-2]+7
```

with the gcd check

$$\begin{aligned} & 2 * \check{i} + 2 = 2 * \hat{i} - 2 \\ \Leftrightarrow\ & 2 * \check{i} - 2 * \hat{i} = -4 \end{aligned}$$

We see that $\gcd(2, 2)$ divides -4. Thus, there could be a dependency. We read off by dividing both sides by the gcd 2 that $\check{i} - \hat{i} = -2$. This could be

1. A true dependency δ^t (read after write) of the form $\check{i} + 2 = \hat{i}$, i.e., with dependency distance 2. "The array position that we read at iteration $\hat{i}$ is the same one that we have written 2 iterations before at $\check{i}$."
   This dependency survives the check for loop bounds and is a real dependency.

2. And/or an anti-dependency δ^a (write after read) of the form $\hat{i} - 2 = \check{i}$. "The array position that we write at iteration $\check{i}$ is the same one that we have read 2 iterations before at $\hat{i}$." This dependency is incompatible with the constraint that $\hat{i} < \check{i}$, because we need data to use from previous iterations before we can use it for subsequent definitions.

More conservatively, one can consider all resulting dependencies as dependencies, without checking for feasibility with the loop bounds.

More precise but computationally more involved algorithms exist, for instance, Fourier-Motzkin elimination for linear systems of inequalities.

Quiz

1. What changes with 4-way associative cache instead of a directly mapped cache? Give an example showing whether the cache miss rates are better or whether they can still be as bad as this lecture showed.
2. Why is the data dependency vector only defined for uniform array access? Is that an oversight?
3. Write down an explicit optimization (for common cases) that makes use of SSE3 instructions to optimize loops. Which side conditions do you need to check? How can you make them easier compared to the general case?
4. How do you recognize perfectly nested loops by analysis of your intermediate language?
5. Should your intermediate language representation have the for loop as a primitive? Should it have the while loop as a primitive? Should it insist on (conditional) gotos as the only way to represent looping behavior? Discuss benefits and disadvantages for various phases of your compiler.
6. Which of the dependencies (all 4 combinations of read/write after write/read) does your compiler have to worry about and for which purpose?
7. Should compiler writers try to convince chip designers to produce fully associative caches to make loop cache optimizations easier?
8. Why did the dependency distance vector flip by swapping loops. It's still the same dependency, right? Why should the vector be different after a swap?
9. How can you implement the gcd test efficiently and which approximations make most sense computationally?

References

[App98] Andrew W. Appel. *Modern Compiler Implementation in ML*. Cambridge University Press, Cambridge, England, 1998.