Memory Hierarchy Optimizations

An Example Memory Hierarchy

Caches: A Quick Review

Optimizing Cache Performance

Two Things We Can Manipulate

Time: Reordering Computation
Space: Changing Data Layout

- What do we know about an object’s location?
  - scalars, structures, pointer-based data
  - structures, arrays, code, etc.

- How can we tell what a better layout would be?
  - how many can we create?

- To what extent can we safely alter the layout?

Types of Objects to Consider

- Scalars
- Structures & Pointers
- Arrays

Types of Objects to Consider

- Scalars
- Structures & Pointers
- Arrays

Scalars

- Locals
- Globals
- Procedure arguments

  - int x;
  - double y;
  - foo(int a){
    int i;
    ...
    x = a*i;
  }

- Is cache performance a concern here?
- If so, what can be done?

Structures and Pointers

- What can we do here?
  - within a node
  - across nodes

  - struct {
    int count;
    double velocity;
    double inertia;
    struct node *neighbors[N];
  } node;

- What limits the compiler’s ability to optimize here?

Arrays

- double A[M][M], B[M][M];
- for i = 0 to M-1
  - for j = 0 to M-1
    - A[i][j] = B[j][i];

- usually accessed within loops nests
  - makes it easy to understand “time”

- what we know about array element addresses:
  - start of array?
  - relative position within array

Data Dependence in Loops

- Dependence can flow across iterations of the loop.
- Dependence information is annotated with iteration information.

  - If dependence is across iterations it is loop carried otherwise loop independent.

  - for (i=0; i<n; i++) {
    A[i] = B[i];
    B[i+1] = A[i];
  }

- What can we do here?
Data Dependence in Loops

- Dependence can flow across iterations of the loop.
- Dependence information is annotated with iteration information.
- If dependence is across iterations it is loop carried otherwise loop independent.

```c
for (i=0; i<n; i++) {
    A[i] = B[i];
    B[i+1] = A[i];
}
```

δ f loop independent

δ f loop carried

Unroll Loop to Find Dependencies

```c
for (i=0; i<n; i++) {
    A[i] = B[i];
    B[i+1] = A[i];
}
```

δ f loop independent

A[0] = B[0];
B[1] = A[0];

Distance/Direction of the dependence is also important.

Distance Vector

```c
for (i=0; i<n; i++) {
    A[i] = B[i];
    B[i+1] = A[i];
}
```

```c
for (j=0; j<m; j++) {
    A[i,j] = ;
    B[i,j+1] = ;
    C[i+1,j] = ;
}
```

A[0,0] = A[0,0]
B[0,1] = B[0,0]
C[1,0] = C[0,1]
A[0,1] = A[0,1]
B[0,2] = B[0,1]
C[1,1] = C[0,2]
A[0,2] = A[0,2]
B[0,3] = B[0,2]
C[1,2] = C[0,3]
A[1,0] = A[1,0]
B[1,1] = B[1,0]
C[2,0] = C[1,1]
A[1,1] = A[1,1]
B[1,2] = B[1,1]
C[2,1] = C[1,2]
A[1,2] = A[1,2]
B[1,3] = B[1,2]
C[2,2] = C[1,3]
B[2,1] = B[2,0]
C[3,0] = C[2,1]
C[3,1] = C[2,2]
C[3,2] = C[2,3]
```

Distance vector is the difference between the target and source iterations.

\[ d = i - l_i \]

Exactly the distance of the dependence, i.e.,

\[ i + d = l_i \]
**Handy Representation:** "Iteration Space"

for $i = 0$ to $N - 1$
for $j = 0$ to $N - 1$
    $A[i][j] = B[j][i]$;

- each position represents an iteration

**Visitation Order in Iteration Space**

for $i = 0$ to $N - 1$
for $j = 0$ to $N - 1$
    $A[i][j] = B[j][i]$;

- Note: iteration space is not data space

**When Do Cache Misses Occur?**

for $i = 0$ to $N - 1$
for $j = 0$ to $N - 1$
    $A[i][j] = B[j][i]$;

**When Do Cache Misses Occur?**

for $i = 0$ to $N - 1$
for $j = 0$ to $N - 1$
    $A[i+j][0] = i*j$;

**Scalar Replacement**

- Replaces subscripted array references with scalars.
- AKA: register pipelining
- Benefits:
  - Improved DS performance
  - Register allocation made possible
  - Easier to software pipeline

**Example: MM**

```c
for (i=0; i<N; i++)
for (j=0; j<N; j++)
for (k=0; k<N; k++)
    C[i][j] = C[i][j] + A[i][k]*B[k][j];
```

- replace $C[][]$ with scalar in inner loop.
- Reduces memory references by $2(N^3-N^2)$
Scalar Replacement data structures

- Let's consider loops without conditionals
- Define the period of a loop carried dependence for edge e, p(e), as the CONSTANT number of iterations between the references at tail and head. (If not constant we can't do it).
- Build a partial dependence graph including:
  - flow (R after W) and
  - input dependencies (R after R)
- And the dependencies:
  - have a constant period
  - are:
    - loop independent or
    - carried by innermost loop

Scalar Replacement Alg

- For a period of p(e) cycles, use p(e)+1 temporaries t0 to tp(e)
- In body of loop:
  - Replace A[i] with t0
  - Replace A[i+j] with tj
- At end of innermost loop body add assignments
  - t0 = tj
- Init temps by peeling off p(e) iterations

Example: MM

```c
for (i=0; i<N; i++)
  for (j=0; j<N; j++)
  for (k=0; k<N; k++)
    C[i][j] = C[i][j] + A[i][k]*B[k][j];
```

- replace C[i][j] with scalar in inner loop.
- Reduces memory references by 2(N^3 - N^2)

Scalar Replacement: Loop Body

```c
for (i=0; i<n; i++)
  b[i+1] = b[i] + f
  a[i] = 2*b[i] + c[i]
```

- We need two temporaries: t0, t1
- Replace b[i] with t0 and b[i+1] with t1
- Insert copies at bottom of loop

Scalar Replacement: Init

```c
for (i=0; i<n; i++)
  t0 = b[0] + f
  t1 = 2*t0 + c[0]
```

1) Peel of p(e) iterations of loop

```c
b[1] = b[0] + f
a[0] = 2*b[0] + c[0]
```

2) after replacement

```c
t0 = b[0]
t1 = t0 + f
a[0] = 2*t0 + c[0]
```

3) If we aren't sure of trip count

```c
if (i>0) {
  t0 = b[0]
t1 = t0 + f
a[1] = t1
a[0] = 2*t0 + c[0]
}
```

Finished

```c
if (n>0) {
  b[1] = b[0] + f
}
```
Optimizing the Cache Behavior of Array Accesses

- We need to answer the following questions:
  - When do cache misses occur?
  - Use "locality analysis".
  - Can we change the order of the iterations (or possibly data layout) to produce better behavior?
  - Evaluate the cost of various alternatives.
  - Does the new ordering/layout still produce correct results?
  - Use "dependence analysis".

Examples of Loop Transformations

- Loop Interchange
- Cache Blocking
- Skewing
- Loop Reversal
- ...

(We will briefly discuss the first two)

Loop Interchange

For $i = 0$ to $N-1$

For $j = 0$ to $N-1$

$A[j][i] = i*j$;

- (assuming $N$ is large relative to cache size)

Cache Blocking (aka "Tiling")

For $i = 0$ to $N-1$

For $j = 0$ to $N-1$

$f(A[i],A[j])$;

For JJ = 0 to $N-1$ by $B$

For $i = 0$ to $N-1$

For $j = JJ$ to max($N-1$, $JJ+B-1$)

$f(A[i],A[j])$;

- Now we can exploit temporal locality

Impact on Visitation Order in Iteration Space

For $i = 0$ to $N-1$

For $j = 0$ to $N-1$

$f(A[i],A[j])$;

For JJ = 0 to $N-1$ by $B$

For $i = 0$ to $N-1$

For $j = JJ$ to max($N-1$, $JJ+B-1$)

$f(A[i],A[j])$;

- Brings square sub-blocks of matrix "b" into the cache.
- Completely uses them up before moving on.

Cache Blocking in Two Dimensions

For $i = 0$ to $N-1$

For $j = 0$ to $N-1$

For $k = 0$ to $N-1$

$c[i,k] += a[i,j]*b[j,k]$;

For JJ = 0 to $N-1$ by $B$

For KK = 0 to $N-1$ by $B$

For $i = 0$ to $N-1$

For $j = JJ$ to max($N-1$, $JJ+B-1$)

For $k = KK$ to max($N-1$, $KK+B-1$)

$c[i,k] += a[i,j]*b[j,k]$;

- Brings square sub-blocks of matrix "b" into the cache.
- Completely uses them up before moving on.
Predicting Cache Behavior through "Locality Analysis"

- Definitions:
  - Reuse: accessing a location that has been accessed in the past
  - Locality: accessing a location that is now found in the cache

- Key Insights
  - Locality only occurs when there is reuse!
  - BUT, reuse does not necessarily result in locality.
    - why not?

Steps in Locality Analysis

1. Find data reuse
   - if caches were infinitely large, we would be finished

2. Determine "localized iteration space"
   - set of inner loops where the data accessed by an iteration is expected to fit within the cache

3. Find data locality:
   - reuse $\&$ localized iteration space $\Rightarrow$ locality

Types of Data Reuse/Locality

```
for i = 0 to 2
  for j = 0 to 100
    A[i][j] = B[j][0] + B[j+1][0];
```

Spatial  Temporal  Group