Loop Optimizations

- Loops are important for performance
- Parallelization
  - May need scalar expansion
- Loop peeling
- Loop fusion
- Loop unrolling
- Many more [*CSE 5441 - Introduction to Parallel Computing*]
  - Loop permutation (interchange)
  - Loop distribution (fission)
  - Loop tiling
  - Loop skewing
  - Index set splitting (generalization of peeling)

Loop Parallelization

- When all iterations of a loop are independent of each other

```
for ( i = 0 ; i < 4096 ; i++ )
    c[i] = a[i] + b[i];
```

- Needs some form of loop dependence analysis, which often involves reasoning about arrays
- May require enabling pre-transformations to make it parallel (e.g., scalar expansion)

```
for ( i = 0 ; i < 4096 ; i++ ) {
    t = a[i] + b[i];
    c[i] = t*t; }
```

Scalar expansion example

```
double tx[4096];
for ( i = 0 ; i < 4096 ; i++ ) {
    tx[i] = a[i] + b[i];
    c[i] = tx[i]*tx[i]; }
t = tx[4095];
```

Loop Peeling

- Goal: extract the first (or last) iteration
  - E.g. wraparound variables for cylindrical coordinates

```
j = N;
for ( i = 0 ; i < N ; i++ ) {
   b[i] = (a[i] + a[j]) / 2;
   j = i; } // assume  j is not live here
```

```
b[0] = (a[0] + a[N]) / 2;
b[1] = (a[1] + a[0]) / 2;
b[2] = (a[2] + a[1]) / 2
…
```

- Peel off the first iteration, then do induction variable analysis and copy propagation

```
j = N;
if (N>=1) {
  b[0] = (a[0] + a[j]) / 2;
  j = 0;
  for ( i = 1 ; i < N ; i++ ) {
     b[i] = (a[i] + a[j]) / 2;
     j = i; } }
```

$\Rightarrow$

```
if (N>=1) {
  b[0] = (a[0] + a[N]) / 2;
  for ( i = 1 ; i < N ; i++ ) {
     b[i] = (a[i] + a[i-1]) / 2; } }
// now we can do unrolling
```

Loop Fusion

- Merge two loops with compatible bounds

```
for ( i = 0 ; i < N ; i++ )
    c[i] = a[i] + b[i];
for ( j = 0 ; j < N ; j++)
    d[j] = c[j] * 2;
```

$\Rightarrow$

```
for ( k = 0 ; k < N ; k++ ) {
    c[k] = a[k] + b[k];
    d[k] = c[k] * 2;
}
```

  - Reduces the loop control overhead (i.e., loop exit test)
  - May improve cache use (e.g. the reuse of **c[k]** above) – especially producer/consumer loops [↓capacity misses]
  - Fewer parallelizable loops and increased work per loop: reduces parallelization overhead (cost of spawn/join)
  - If the loop bounds are "±1 off" – use peeling
  - Not always legal – need loop dependence analysis

Loop Unrolling

- Loop unrolling: extend the body

```
for ( i = 0 ; i < 4096 ; i++ )
    c[i] = a[i] + b[i];
```

```
for ( i = 0 ; i < 4095 ; i +=2 ) {
    c[i] = a[i] + b[i];
    c[i+1] = a[i+1] + b[i+1];
} // unroll factor of 2
```

  - Reduces the “control overhead” of the loop: makes the loop exit test (i < 4096) execute less frequently
  - Hardware advantages: instruction-level parallelism; fewer pipeline stalls
  - Issue: loop bound may not be a multiple of unroll factor
  - Problem: high unroll factors may degrade performance due to register pressure and spills

Register Pressure Study

- Thanks to Albert Hartono for these examples
  - trac.mcs.anl.gov/projects/performance/wiki/Orio
- Unrolling for matrix-vector multiplication

```
for ( i=0; i<=N-1; i++ ) {
  for ( j=0; j<=N-1; j++ ) {
    y[i] = y[i] + A[i][j]*x[j];
  }
}
```

```
for ( i=0; i<=N-1; i++ ) {
  for (j=0; j<=N-4; j=j+4 ) {
    y[i]=y[i]+A[i][j]*x[j];
    y[i]=y[i]+A[i][j+1]*x[j+1];
    y[i]=y[i]+A[i][j+2]*x[j+2];
    y[i]=y[i]+A[i][j+3]*x[j+3];
  }
  for ( ; j<=N-1; j=j+1 ) // if N%4 != 0
    y[i]=y[i]+A[i][j]*x[j];
}
```

Unroll the j loop by an unroll factor of 4

Another Unrolled Version

Unroll the i loop by a factor of 4

```
for ( i=0; i<=N-1; i=i+4 ) {
  for ( j=0; j<=N-1; j=j++ ) {
    y[i]=y[i]+A[i][j]*x[j];
  }
  for ( j=0; j<=N-1; j=j++ ) {
    y[i+1]=y[i+1]+A[i+1][j]*x[j];
  }
  for ( j=0; j<=N-1; j=j++ ) {
    y[i+2]=y[i+2]+A[i+2][j]*x[j];
  }
  for ( j=0; j<=N-1; j=j++ ) {
    y[i+3]=y[i+3]+A[i+3][j]*x[j];
  }
} // assume N%4 == 0
```

Fuse the j loops (unroll-and-jam)

```
for ( i=0; i<=N-4; i=i+4 ) {
  for ( j=0; j<=N-1; j++ ) {
    y[i]=y[i]+A[i][j]*x[j];
    y[i+1]=y[i+1]+A[i+1][j]*x[j];
    y[i+2]=y[i+2]+A[i+2][j]*x[j];
    y[i+3]=y[i+3]+A[i+3][j]*x[j];
  }
}
```

Unroll Both Loops

Unroll the i loop by a factor of 2 and the j loop by a factor of 2 and then fuse the j loops

```
for ( i=0; i<=N-2; i=i+2 ) {
  for ( j=0; j<=N-2; j=j+2 ) {
    y[i]=y[i]+A[i][j]*x[j];
    y[i]=y[i]+A[i][j+1]*x[j+1];
    y[i+1]=y[i+1]+A[i+1][j]*x[j];
    y[i+1]=y[i+1]+A[i+1][j+1]*x[j+1];
  }
}
```

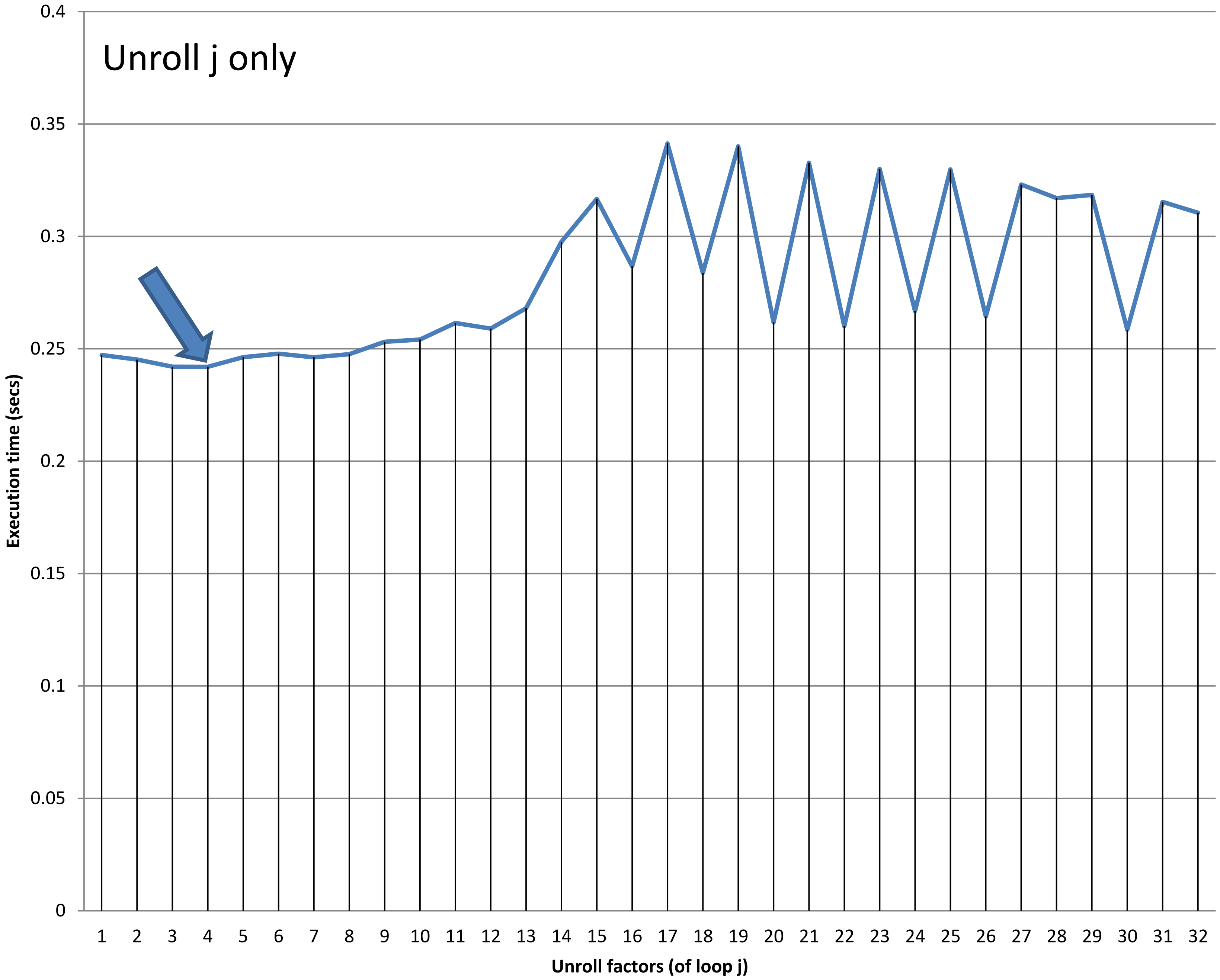
Scalar Replacement

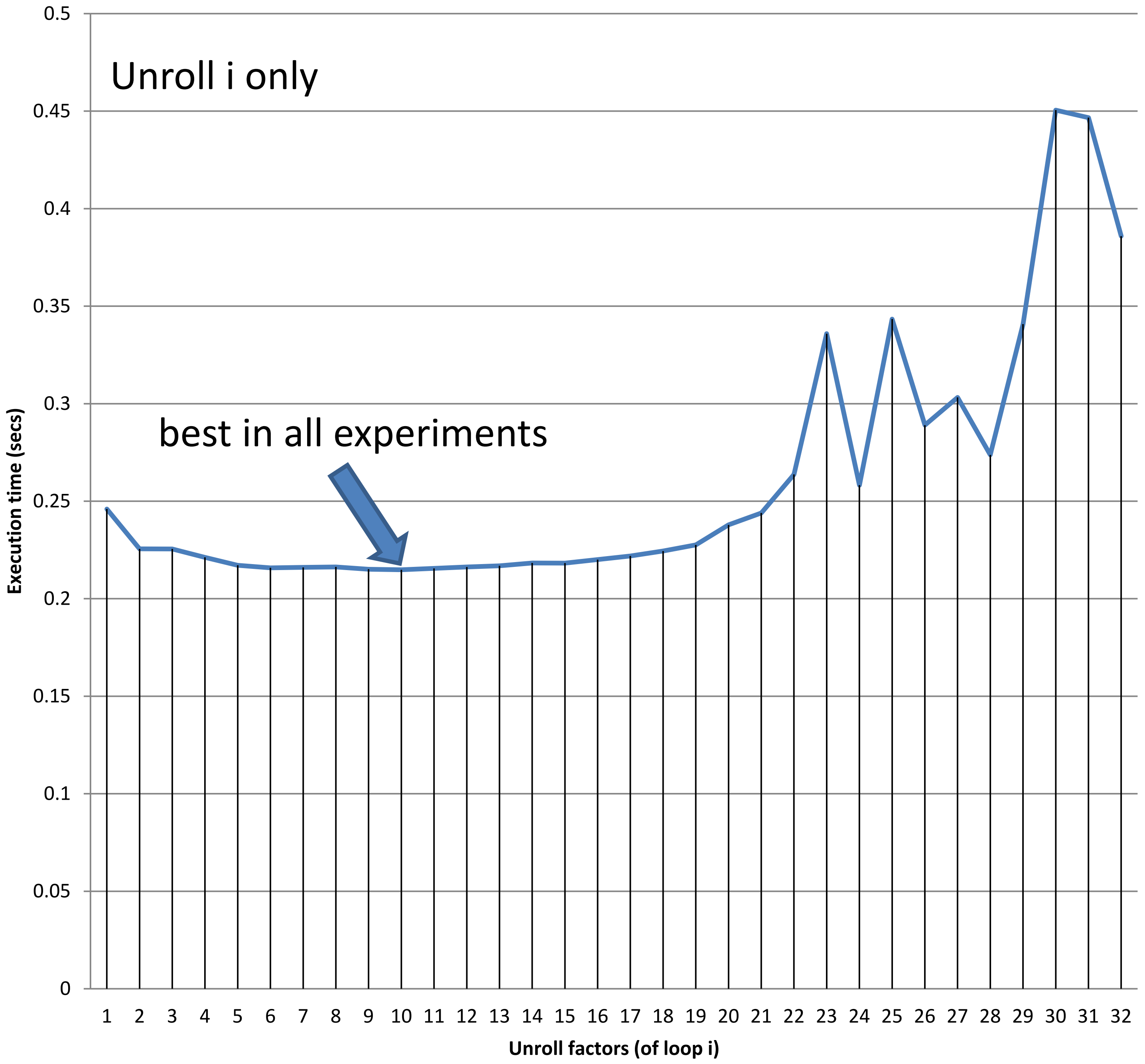
Replace array references with scalars

```
for ( i=0; i<=N-2; i=i+2 ) {
  double scv_3, scv_4;
  scv_3 = y[i]; scv_4 = y[i+1];
  for ( j=0; j<=N-2; j=j+2 ) {
    double scv_1, scv_2;
    scv_1 = x[j]; scv_2 = x[j+1];
    scv_3=scv_3+A[i][j]*scv_1;
    scv_3=scv_3+A[i][j+1]*scv_2;
    scv_4=scv_4+A[i+1][j]*scv_1;
    scv_4=scv_4+A[i+1][j+1]*scv_2;
  }
  y[i] = scv_3; y[i+1] = scv_4;
}
```

Experimental Setup

- N=10000
- Scalar replacement used in all experiments
- gcc 4.2.4 with -O3 optimization flag
- Multi-core Intel Xeon workstation
  - dual quad-core E5462 Xeon processors (8 cores total) running at 2.8 GHz (1600 MHz FSB), 32 KB L1 cache, 12 MB of L2 cache, 16 GB of DDR2 RAM
- All combination of unroll factors for i and j: values of 1,2,3,...,32 (total: 1024 versions)
- The Orio tool determined that unroll factor 10 for i and unroll factor 1 for j is the best


Unroll j only
Execution time (secs)
0.4
0.35
0.3
0.25
0.2
0.15
0.1
0.05
0
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32
Unroll factors (of loop j)


Unroll i only
best in all experiments
Execution time (secs)
0.5
0.45
0.4
0.35
0.3
0.25
0.2
0.15
0.1
0.05
0
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32
Unroll factors (of loop i)