Tuning Programs For Performance

- Don’t tune your program unless there is a performance issue
- Use performance instrumentation tools:

  prof: performance profiler based on sampling

  gprof: extension to prof with call graphs

  pixie: instruction execution counts (does not exist on SPARC)

  Read their man pages for more information

- Example:

cc -p foo.c -o foo	(compile with -p for using “prof”)
foo	(execute foo will generate a file mon.out)
prof foo	(get profiling information)

- Rule of thumb: 10% of your program takes 90% of the time
- Indentify the “hot” spots of your program and only tune the hot spots

Trade Space for Time

- Data structure augmentation

  Extra information to help access data structure efficiently

  Example: “footer” for the memory allocation can be used to find the beginning of the previous memory

  Data structure for “fast paths”

- **Store precomputed results**

  Avoid repetitive computations by storing results in data structures

  e.g. tables of trignometric values, shading tables in graphics ...

- **Caching**

  Data accessed most often should be the cheapest to access

  Example: file systems cache recently accessed data in memory

- **Lazy evaluation**

  Don’t evaluate an item until it is needed

  “Never do today what you can put off until tomorrow.”

Trading Time for Space

- Packing

  Dense storage representation can decrease storage cost by increasing the time required to store and retrieve.

  Example: compressed file system on disks

- **Interpreters**

  Represent operations compactly and then interpret them later

  Example: postscript for printing

- **Opposite of "trading space for time"**

  Data structure reduction

  Recompute results

  Uncaching

  Eager evaluation

Loop Rules

- The code you are modifying is probably a loop
- Need to know whether your compiler does it for you before doing it
- Rule 1: Code motion out of loops (some compilers do this)

Example

```
for ( i = 0; i < n; i++ )
    x[i] = x[i] * exp( sqrt( Pi/2));

factor = exp( sqrt( Pi/2));
for ( i = 0; i < n; i++ )
    x[i] = x[i] * factor;
```

- **Rule 2: Combining tests (no compilers do this)**

Example

```
for ( i = 0; i < n && x[i] != y; i++ );
if ( i < n ) ...                          /* found */

x[n] = y;                                 /* add sentinel */
for ( i = 0; x[i] != y; i++ );
if ( i < n ) ...                          /* found */
```

More on Loop Rules

- **Rule 3: Loop unrolling (some compilers do this, but very few)**

  Example: sum up the elements in a vector

```
sum = 0;
for ( i = 0; i < 8; i++ )
    sum += x[i];
```

↓

```
sum = x[0]+x[1]+x[2]+x[3]+x[4]+x[5]+x[6]+x[7];
```

  What if looping n times?

- **Rule 4: Loop fusion (almost no compilers do this)**

  If two nearby loops iterate the same number of times, combine them

- **Remember**

  Don’t sacrifice readability for performance unless absolutely necessary

  When a new system is fast enough, you may want to use old, readable code!

Logic Reasoning

- Rule 1: Exploit algebraic identities

```
if ( sqrt( x ) > 0 ) ...
```

↓

```
if ( x > 0 ) ...
```

- **Rule 2: Short-circuiting monotone functions**

```
sum = 0;
for ( i = 0; i < n; i++ )
     sum += x[i];
if ( sum > CutOff ) ...
```

↓

```
sum = 0;
for ( i = 0; i < n && sum <= CutOff; i++ )
     sum += x[i];
if ( sum > cutOff ) ...
```

Logic Reasoning, cont'd

- Rule 3: reordering tests to test inexpensive and common conditions first

```
if ( ComplexTest( x, y ) && !z ) ...

if ( !z && ComplexTest( x, y ) ) ...
```

- **Rule 4: precompute logical functions; store results in a lookup table**

```
int CharType( int c ) {
    return typeTable[ c ];
}
```

It can be a macro if needed.

- **Rule 5: boolean variable elimination**

```
v = LogicalExpression();
S1;
if ( v )
    s2;
else
    s3;
```

```
if ( LogicalExpression() ) {
    s1;
    s2;
}
else {
    s1;
    s2;
}
```

Procedure Rules

- **Rule 1: collapsing procedure hierarchies**

  Inline “hot” procedure calls

- **Rule 2: Exploit common cases**

  Handle all cases correctly and common cases efficiently

- **Rule 3: Transformations on recursive calls**

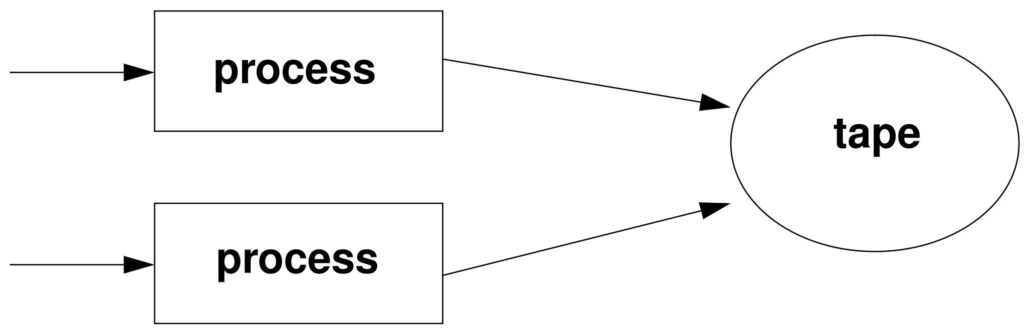
  Make sure write “tail-recursive” calls (some compilers handle tail recursion)

  Transform to loops if necessary

- **Rule 4: Parallelism**

  Asynchronous accesses can be very useful for overlaping CPU and I/O devices

  Example: tape stream mode and prefetching



Expression Rules

- **Rule 1: Compile-time initialization**

```
static int x = 5;
```

- **Rule 2: Exploit algebraic identities**

```
log( a ) + log( b )
```

$\downarrow$

```
log( a * b )
```

- **Rule 3: Common subexpression elimination (many compilers do this)**

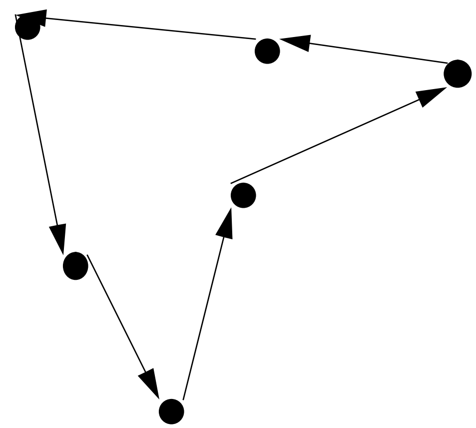
  Store the result of first computation for the second

  (see trading space for time)

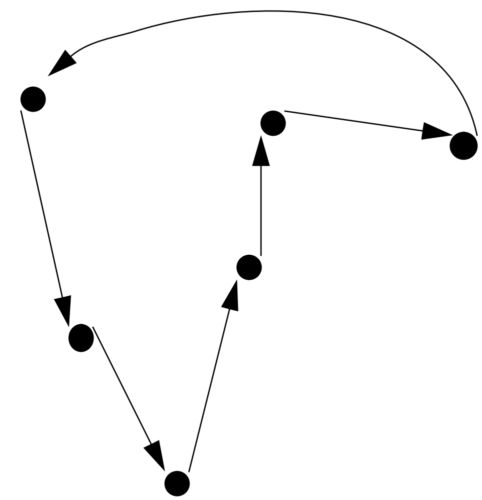
- **Rule 4: Exploit word parallelism**

  Instructions on bytes can be slower on 32-bit or 64-bit words

An Example: Travelling Salesman Problem



Optimal tour



Nearest-neighbor

- Cities are random points in the plane
- Find the shortest tour to travel all cities

Program

- tsp.h:

```
#ifndef TSP_INCLUDED
#define TPS_INCLUDED

#define  MAX_CITIES   5000          /* max number of cities */
#define  MAX          0x0000ffff    /* max dimension */
#define  TRUE  1
#define  FALSE 0

struct Point {                      /* a point is a pair (x, y) */
    double x;
    double y;
};

#endif
```

Main Program, cont'd

```
RandomPoints( int n, struct Point p[] ) {
    int i;
    srandom( 7 );
    for ( i = 0; i < n; i++ )  {
        p[i].x = ( random() % MAX ) / (( double ) MAX );
        p[i].y = ( random() % MAX ) / (( double ) MAX );
    }
}

int GetTime( void ) {
    int msecs; struct timeval tv; struct timezone zone;
    gettimeofday( &tv, &zone );
    msecs = 1000 * tv.tv_sec + tv.tv_usec / 1000;
    return msecs;
}

void main( void )
{
    int n, tour[ MAX_CITIES ], t1, t2;
    struct Point p[ MAX_CITIES ];
    printf( "Number of cities: " );
    scanf( "%d", &n );
    RandomPoints( n, p );              /* initialize all the points */
    t1 = GetTime();
    ApproxTSTour( n, p, tour );        /* find a good tour */
    t2 = GetTime();
    printf( "%d, Time = %d ms\n", n, t2 - t1 );
}
```

Original Version

```
double Distance( struct Point p, struct Point q ) {
    return sqrt((p.x-q.x)*(p.x-q.x)+(p.y-q.y)*(p.y-q.y));
}

void ApproxTSTour( int n, struct Point p[], int tour[] ){
    int current, visited[ MAX_CITIES ], i, j, jMin;
    double dMin;
    for ( i = 0; i < n; i++ )
        visited[i] = FALSE;
    current = 0;
    visited[current] = TRUE;
    tour[0] = current;
    for ( i = 1; i < n; i++ ) {
        dMin = MAX;
        for ( j = 0; j < n; j++ ) {
            if ( !visited[j]
                && ( Distance( p[current], p[j] ) < dMin ) ) {
                jMin = j;
                dMin = Distance( p[current], p[j] );
            }
        }
        current = jMin;
        visited[current] = TRUE;
        tour[i] = current;
    }
}
```

Store Computed Result

```
double Distance( struct Point p, struct Point q ) {
    return sqrt((p.x-q.x)*(p.x-q.x)+(p.y-q.y)*(p.y-q.y));
}

void ApproxTSTour( int n, struct Point p[], int tour[] ){
    int current, visited[ MAX_CITIES ], i, j, jMin;
    double dMin, dj;
    for ( i = 0; i < n; i++ )
        visited[i] = FALSE;
    current = 0;
    visited[current] = TRUE;
    tour[0] = current;
    for ( i = 1; i < n; i++ ) {
        dMin = MAX;
        for ( j = 0; j < n; j++ ) {
            if ( !visited[j]
                && ((dj = Distance( p[current], p[j])) < dMin ) ) {
                jMin = j;
                dMin = dj;
            }
        }
        current = jMin;
        visited[current] = TRUE;
        tour[i] = current;
    }
}
```

Compute Psuedo Distance

```
double Distance( struct Point p, struct Point q ) {
    return (p.x-q.x)*(p.x-q.x)+(p.y-q.y)*(p.y-q.y);
}

void ApproxTSTour( int n, struct Point p[], int tour[] ) {
    int current, visited[ MAX_CITIES ], i, j, jMin;
    double dMin, dj;
    for ( i = 0; i < n; i++ )
        visited[i] = FALSE;
    current = 0;
    visited[current] = TRUE;
    tour[0] = current;
    for ( i = 1; i < n; i++ ) {
        dMin = MAX;
        for ( j = 0; j < n; j++ ) {
            if ( !visited[j]
                && ((dj = Distance( p[current], p[j])) < dMin ) ) {
                jMin = j;
                dMin = dj;
            }
        }
        current = jMin;
        visited[current] = TRUE;
        tour[i] = current;
    }
}
```

Visit Unvisited Cities

```
double Distance( struct Point p, struct Point q ) {
    return (p.x-q.x)*(p.x-q.x)+(p.y-q.y)*(p.y-q.y);
}

void ApproxTSTour( int n, struct Point p[], int tour[] )
{
    int current, i, j, jMin, high;
    double dMin, dj;

    for ( i = 0; i < n; i++ )
        tour[i] = i;


    high = n-1;
    while ( high > 0 ) {
        dMin = MAX;
        for ( j = 0; j < high; j++ ) {
            dj = Distance( p[tour[high]], p[tour[j]] );
            if ( dj  < dMin ) {
                jMin = j;
                dMin = dj;
            }
        }
        current = tour[jMin];
        tour[jMin] = tour[high];
        tour[high] = current;
        high--;
    }
}
```

Inline Procedure Call

```
#define Distance(p,q) ((p.x-q.x)*(p.x-q.x)+(p.y-q.y)*(p.y-q.y))

void ApproxTSTour( int n, struct Point p[], int tour[] )
{
    int current, i, j, jMin, high;
    double dMin, dj;
    for ( i = 0; i < n; i++ )
        tour[i] = i;
    current = tour[ n ];
    high = n-1;
    while ( high > 0 ) {
        dMin = MAX;
        for ( j = 0; j < high; j++ ) {
            dj = Distance( p[current], p[tour[j]] );
            if ( dj  < dMin ) {
                jMin = j;
                dMin = dj;
            }
        }
        current = tour[jMin];
        tour[jMin] = tour[high];
        tour[high] = current;
        high--;
    }
}
```

Rearrange Tests

```
#define XDistance(p,q) ((p.x-q.x)*(p.x-q.x))
#define YDistance(p,q) ((p.y-q.y)*(p.y-q.y))

void ApproxTSTour( int n, struct Point p[], int tour[] )
{
    int current, i, j, jMin, high;
    double dMin, dj;
    for ( i = 0; i < n; i++ )
        tour[i] = i;
    current = tour[ n ];
    for ( high = n-1; high > 0; high-- ) {
        dMin = MAX;
        for ( j = 0; j < high; j++ ) {
             if ( ( dx = XDistance( p[current], p[tour[j]] ) ) < dMin ) {
                dj = dx + YDistance( p[current], p[tour[j]] );
                if ( dj  < dMin ) {
                    jMin = j;
                    dMin = dj;
                }
            }
         }
        current = tour[jMin];
        tour[jMin] = tour[high];
        tour[high] = current;
     }
}
```

Performance Results

SUN SPARC (40Mhz) using lcc

Programs	Time (ms)	Speedup
Original	14,152	1.00
Store Computed Results	13,923	1.02
Compute Psuedo Distance	4,632	3.06
Visit Unvisited Cities	3,713	3.81
Inline Procedure Call	2,052	6.90
Rearrange Tests	1,494	9.47

Performance Results

SGI Indigo (100Mhz MIPS R4000) using cc -O

Programs	Time (ms)	Speedup
Original	5,988	1.00
Store Computed Results	5,999	1.00
Compute Psuedo Distance	1,387	4.32
Visit Unvisited Cities	1,184	5.06
Inline Procedure Call	564	10.62
Rearrange Tests	508	11.79

Performance Results

DEC Alpha 3000/400 (150Mhz Alpha) using cc -O3

Programs	Time (ms)	Speedup
Original	2,103	1.00
Store Computed Results	1,744	1.21
Compute Psuedo Distance	707	2.97
Visit Unvisited Cities	588	3.57
Inline Procedure Call	410	5.12
Rearrange Tests	363	5.79