PERFORMANCE ENGINEERING
OF SOFTWARE SYSTEMS

Basic Performance Engineering

Saman Amarasinghe

Fall 2010

Basic Performance Engineering

Maximum use of the compiler/processor/system — Matrix Multiply Example

Modifying data structures — Today

Modifying code structures — Today

Using the right algorithm — Most Bithacks

© Saman Amarasinghe 2009

Bentley's Rules

There is no "theory" of performance programming

Performance Programming is:

- Knowledge of all the layers involved
- Experience in knowing when and how performance can be a problem
- Skill in detecting and zooming in on the problems
- A good dose of common sense

A set of rules

- Patterns that occur regularly
- Mistakes many make
- Possibility of substantial performance impact
- Similar to "Design Patterns" you learned in 6.005

© Saman Amarasinghe 2009

Bentley's Rules

A. **Modifying Data**
B. **Modifying Code**

© Saman Amarasinghe 2009

Bentley's Rules

A. Modifying Data

1. Space for Time
2. Time for Space
3. Space and Time

B. Modifying Code

© Saman Amarasinghe 2009

Bentley's Rules

A. Modifying Data

1. Space for Time
   a. Data Structure Augmentation
   b. Storing Precomputed Results
   c. Caching
   d. Lazy Evaluation
2. Time for Space
3. Space and Time

B. Modifying Code

© Saman Amarasinghe 2009

Data Structure Augmentation

Add some more info to the data structures to make common operations quicker

When is this viable?

- Additional information offers a clear benefit
- Calculating the information is cheap/easy
- Keeping the information current is not too difficult

Examples?

- Faster Navigation
  - Doubly linked list and Delete Operation
- Reduced Computation
  - Reference Counting

© Saman Amarasinghe 2009

Storing Precomputed Results

Store the results of a previous calculation. Reuse the precomputed results than redoing the calculation.

When is this viable?

- Function is expensive
- Function is heavily used
- Argument space is small
- Results only depend on the arguments
- Function has no side effects
- Function is deterministic

Examples:

© Saman Amarasinghe 2009

Precomputing Template Code

```
result precompute[MAXARG];

result func_initialize(int arg)
{
    for(i=0; i < MAXARG; i++)
        precompute[arg] = func(arg);
}

result func_apply(int arg)
{
    return precompute[arg];
}
```

© Saman Amarasinghe 2009

Pascal's Triangle

```
int pascal(int y, int x)
 {
  if(x == 0) return 1;
  if(x == y) return 1;
  return pascal(y-1, x-1) + pascal(y-1, x);
 }
```

Normal

```
 int pt[MAXPT][MAXPT];

 main() {
 …
  for(i=0; i < PTMAX; i++) {
    pt[i][0] = 1;
    pt[i][i] = 1;
    for(j=1; j < i; j++)
      pt[i][j] = pt[i-1][j-1] + pt[i-1][j];
 }

int pascal(int y, int x) {
     return pt[y][x];
}
```

Precomputation

$$
\begin{array}{ccccccccc}
 & & & & 1 & & & & \\
 & & & 1 & & 1 & & & \\
 & & 1 & & 2 & & 1 & & \\
 & 1 & & 3 & & 3 & & 1 & \\
1 & & 4 & & 6 & & 4 & & 1
\end{array}
$$

© Saman Amarasinghe 2009

Another example of precomputing

```
unsigned long fib(int n)
{
    if(n==1) return 1;
    if(n==2) return 1;
    return fib(n-1) + fib(n-2);
}
```

```
unsigned long fib(int n)
{
    int i;
    unsigned long prev, curr, tmp;
    if(n==1) return 1;
    if(n==2) return 1;
    prev = 1;
    curr = 1;
    for(i=3; i <=n; i++ ) {
        tmp = prev + curr;
        prev = curr;
        curr = tmp;
    }
    return curr;
}
```

© Saman Amarasinghe 2009

Caching

Store some of the heavily used/recently used results so they don't need to be computed

When is this viable?

- Function is expensive
- Function is heavily used
- Argument space is large
- There is temporal locality in accessing the arguments
- A single hash value can be calculated from the arguments
- There exists a "good" hash function
- Results only depend on the arguments
- Function has no side effects
- Coherence
  - Is required:
    - Ability to invalidate the cache when the results change
    - Function is deterministic
  - Or stale data can be tolerated for a little while

© Saman Amarasinghe 2009

Caching Template Code

```
typedef struct cacheval {
        argtype1 arg1;
        …
        argtypen argn;
        resulttype result;
}
struct cacheval cache[MAXHASH];

resulttype func_driver(argtype1 a1, …, argtypen an) {
        resulttype res;
        int bucket;
        bucket = get_hash(a1, a2, …, an);
        if((cache[bucket].arg1 == a1)&&…&&(cache[bucket].argn == an))
                return cache[bucket].result;
        res = func(a1, …, an);
        cache[bucket].arg1 = a1;
        …
        cache[bucket].argn = an;
        cache[bucket].result = res;
        return res;
}
```

© Saman Amarasinghe 2009

Lazy Evaluation

Differ the computation until the results are really needed

When is this viable?

- Only a few results of a large computation is ever used
- Accessing the result can be done by a function call
- The result values can be calculated incrementally
- All the data needed to calculate the results will remain unchanged or can be packaged-up

© Saman Amarasinghe 2009

Lazy Template Code

```
resulttype precompute[MAXARG];

resulttype func_apply(int arg)
{
    resulttype res;
    if(precompute[arg] != EMPTY)
      return precompute[arg];

    res = func(arg);
    precompute[arg] = res;
    return res;
}
```

© Saman Amarasinghe 2009

Pascal's Triangle

```
int pascal(int y, int x)
{
  if(x == 0) return 1
  if(x == y) return 1;
  return pascal(y-1, x-1) + pascal(y-1, x);
}
```

Normal

$$
\begin{array}{ccccccccc}
 & & & & 1 & & & & \\
 & & & 1 & & 1 & & & \\
 & & 1 & & 2 & & 1 & & \\
 & 1 & & 3 & & 3 & & 1 & \\
1 & & 4 & & 6 & & 4 & & 1
\end{array}
$$

```
int pt[MAXPT][MAXPT];

main() {
...
 for(i=0; i < PTMAX; i++) {
    pt[i][0] = 1;
    pt[i][i] = 1;
    for(j=1; j < i; j++)
      pt[i][j] = pt[i-1][j-1] + pt[i-1][j];
}

int pascal(int y, int x) {
      return pt[y][x];
}
```

Precomputation

```
int pt[MAXPT][MAXPT];

int pascal(int y, int x)
{
  if(x == 0) return 1;
  if(x == y) return 1;
  if(pt[y][x] > 0) return pt[y][x];
  val = pascal(y-1, x-1) + pascal(y-1, x);
  pt[y][x] = val;
  return val;
}
```

Lazy Evaluation

© Saman Amarasinghe 2009

Bentley's Rules

A. Modifying Data

1. Space for Time
2. Time for Space
    1. Packing/Compression
    2. Interpreters
3. Space and Time

B. Modifying Code

© Saman Amarasinghe 2009

Packing/Compression

Reduce the space of the data by storing them as "processed" which will require additional computation to get the data.

When is it viable?

- Storage is at a premium
  - Old days → most of the time!
  - Now
    - Embedded devices with very little memory/storage
    - Very large data sets
- Ability to drastically reduce the data size in storage
- Extraction process is amenable to the required access pattern
  - Batch – expand it all
  - Steam
  - Random access

© Saman Amarasinghe 2009

Packing / Compression

Packing Level

- Packing in memory
- Packing out of core storage

Packing Methods

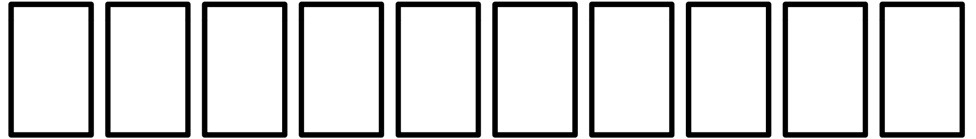
- Use smaller data size
- Eliminate leading zeros
- Eliminate repetitions (LZ77)
- Heavyweight compression

© Saman Amarasinghe 2009

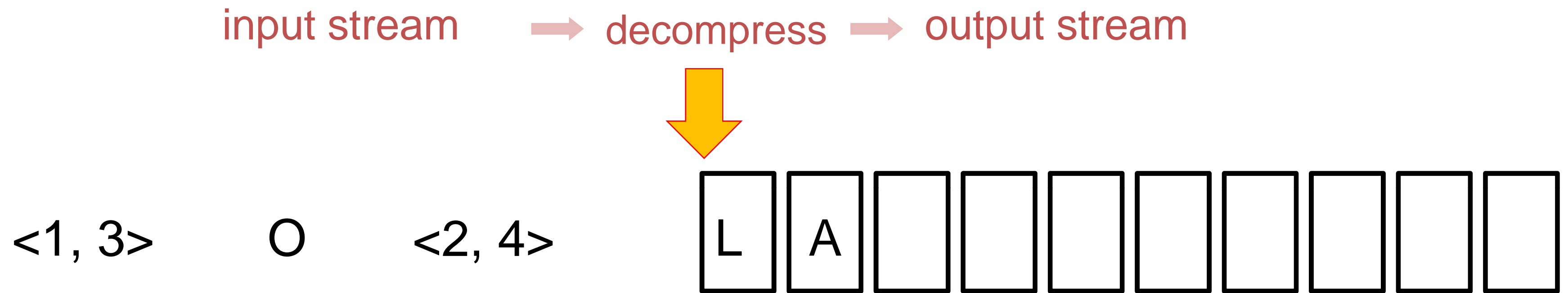
LZ77 Basics

input stream → decompress → output stream

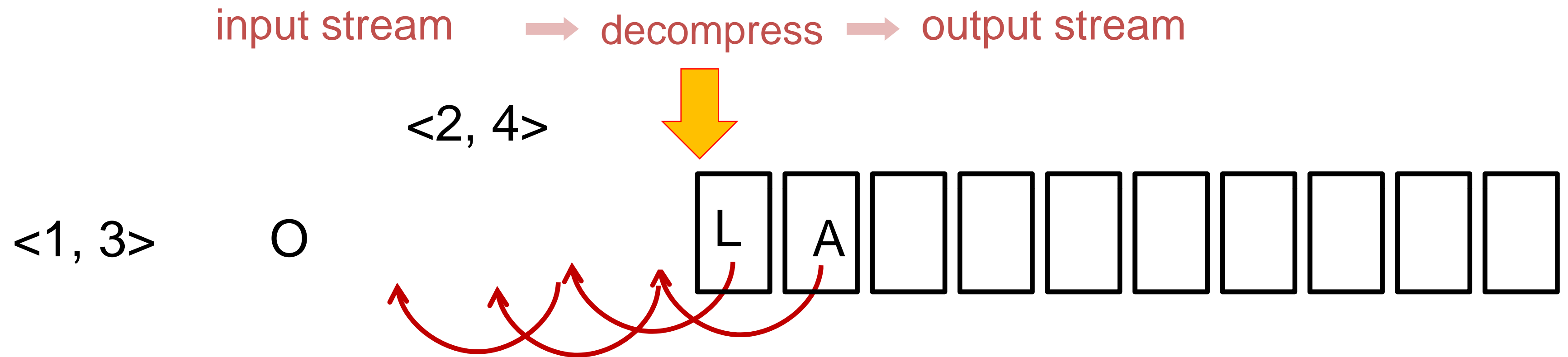
<1, 3>   O   <2, 4>   L   A



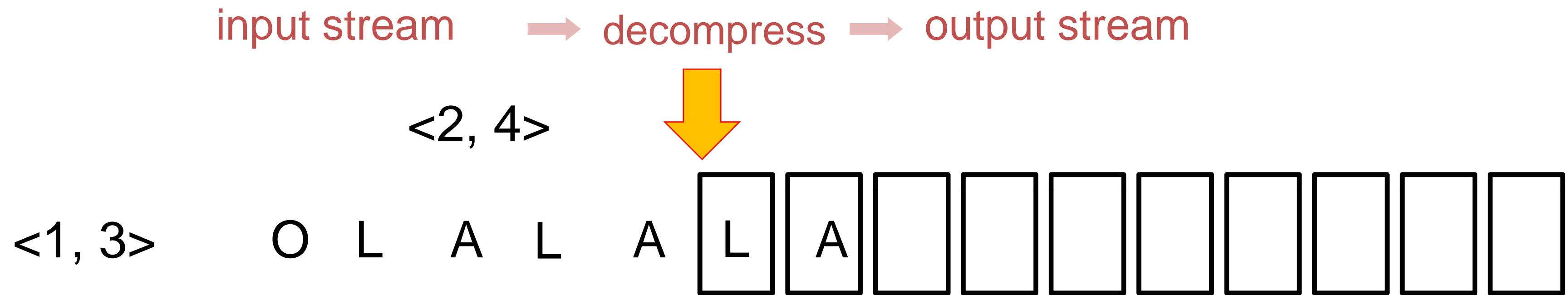
LZ77 Basics



LZ77 Basics



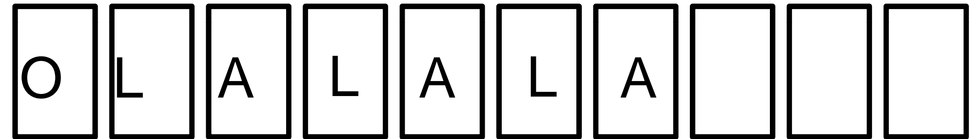
LZ77 Basics



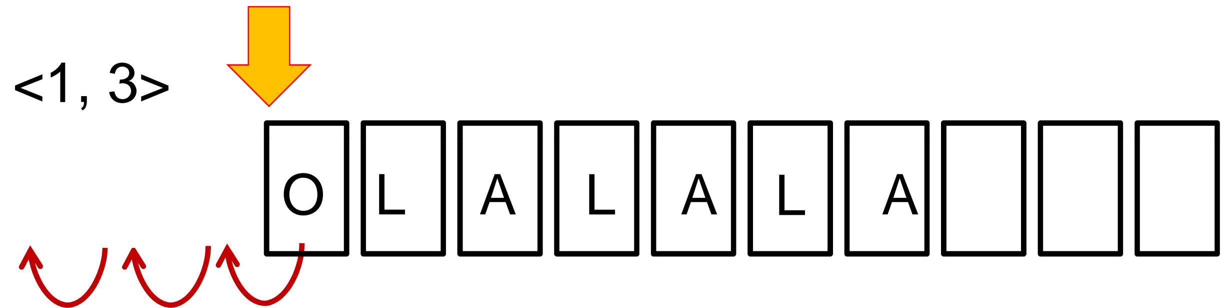
LZ77 Basics

input stream → decompress → output stream

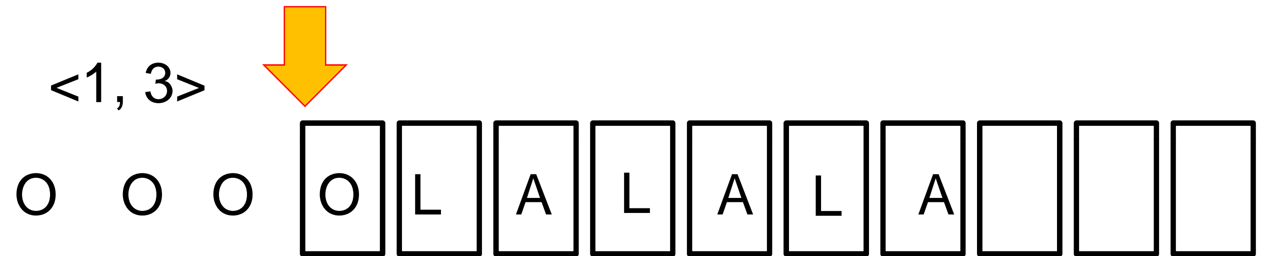
<1, 3>



LZ77 Basics



LZ77 Basics



Interpreters

Instead of writing a program to do a computation, use a language to describe the computation at a high level and write an interpreter for that language

Benefits:

- Nice and clean abstraction of the language
- Easy to add/change operations by changing the HLL program
- Much more compact representation

Examples:

- String processing
- Bytecodes

© Saman Amarasinghe 2009

Bentley's Rules

A. Modifying Data

1. Space for Time
2. Time for Space
3. Space and Time
    1. SIMD

B. Modifying Code

© Saman Amarasinghe 2009

SIMD

Store short width data packed into the machine word (our Intel machines are 64 bit)

- 64 Booleans (unsigned long long)
- 2 32-bit floats
- 2 32-bit integers
- 4 16-bit integers
- 8 8-bit integers

Intel SSE instructions operate on these multi granularities

Single operation on all the data items

Win-win situation both faster and less storage

When viable?

- If the same operation is performed on all the data items
- Items can be stored contiguous in memory
- Common case don't have to pick or operate on each item separately

© Saman Amarasinghe 2009

Example: overlap in battleship boards

```
#define BSZ     64

int overlap(int board1[BSZ][BSZ],
             int board2[BSZ][BSZ])
{
   int i; j;
   for(i=0; i<BSZ; i++)
      for(j=0; j<BSZ; j++)
         if((board1[i][j] == 1)&&
            (board2[i][j] == 1))
               return 1;
   return 0;
}
```

```
#define BSZ     64

int overlap(
   unit64_t board1[BSZ],
   unit64_t board2[BSZ])
{
  int i;
   for(i=0; i<BSZ; i++)
      if((board1[i] & board2[i]) != 0)
         return 1;
   return 0;
}
```

© Saman Amarasinghe 2009

Bentley's Rules

A. Modifying Data

B. Modifying Code

1. Loop Rules
2. Logic Rules
3. Procedure Rules
4. Expression Rules
5. Parallelism Rules

© Saman Amarasinghe 2009

Bentley's Rules

A. Modifying Data

B. Modifying Code

1. Loop Rules
   a. Loop Invariant Code Motion
   b. Sentinel Loop Exit Test
   c. Loop Elimination by Unrolling
   d. Partial Loop Unrolling
   e. Loop fusion
   f. Eliminate wasted iterations
2. Logic Rules
3. Procedure Rules
4. Expression Rules
5. Parallelism Rules

© Saman Amarasinghe 2009

Loops

If each program instruction is only executed once

- 3 GHz machine
- Requres 12 Gbytes of instructions a second! (1 CPI, 32 bit instructions)
- A 100 GB disk full of programs done in 8 seconds

Each program instruction has to run millions of times

→ **Loops**

90% of the program execution time in 10% of the code

- All in inner loops

© Saman Amarasinghe 2009

Loop Invariant Code Motion

Move as much code as possible out of the loops

Compilers do a good job today

- Analyzable code
- Provably results are the same in every iteration
- Provably no side effects

Viability?

- Loop invariant computation that compiler cannot find
- The cost of keeping the value in a register is amortized by the savings

© Saman Amarasinghe 2009

Loop Invariant Code Motion Example

```
for(i=0;  i < N;  i++)
   X[i] = X[i]  * exp(sqrt(PI/2));
```

```
double factor;
factor = exp(sqrt(PI/2));
for(i=0;  i < N;  i++)
   X[i] = X[i]  * factor;
```

© Saman Amarasinghe 2009

Sentinel Loop Exit Test

When we iterate over a data to find a value, we have to check the end of the data as well as for the value.

Add an extra data item at the end that matches the test

Viability?

- Early loop exit condition that can be harnessed as the loop test
- When an extra data item can be added at the end
- Data array is modifiable

© Saman Amarasinghe 2009

Example of Sentinel Loop Exit Test

```
#define DSZ 1024
datatype array[DSZ];

int find(datatype val)
{
    int i;
    for(i=0; i<DSZ; i++)
        if(array[i] == val)
            return i;
    return -1;
OR
   i = 0;
   while((i<DSZ)&&(array[i] != val))
        i++;
    if(i==DSZ) return -1;
    return i;
}
```

```
#define DSZ 1024
datatype array[DSZ+1];

int find(datatype val)
{
    int i;
    array[DSZ] = val;
    i = 0;
    while(array[i] != val)
        i++;

    if(i == DSZ)
            return -1;
    return i;
}
```

© Saman Amarasinghe 2009

Loop Elimination by Unrolling

Known loop bounds → can fully unroll the loop

Viability?

- Small number of iterations (code blow-up is manageable)
- Small loop body (code blow-up is manageable)
- Little work in the loop body (loop test cost is non trivial)

Can get the compiler to do this.

Example:

```
sum = 0;
for(i=0; i<10; i++)
  sum = sum + A[i];
```

```
sum = A[0] + A[1] + A[2] + A[3] +
      A[4] + A[5] + A[6] + A[7] +
      A[8] + A[9];
```

© Saman Amarasinghe 2009

Partial Loop Unrolling

Make a few copies of the loop body.

Viability?

- Work in the loop body is minimal (viable impact of running the loop test fewer number of times)
- Or the ability to perform optimizations on combine loop bodies

Can get the compiler to do this

Example:

```
sum = 0;
for(i=0; i<n; i++)
   sum = sum + A[i];
```

```
sum = 0;
for(i=0; i<n-3; i += 4)
     sum += A[i] + A[i+1] + A[i+2] + A[i+3];
for(; i<n; i++)
     sum = sum + A[i];
```

© Saman Amarasinghe 2009

Loop Fusion

When multiple loops iterate over the same set of data put the computation in one loop body.

Viability?

- No aggregate from one loop is needed in the next
- Loop bodies are manageable

Example

```
amin = INTMAX;
for(i=0; i<n; i++)
   if(A[i] < amin) amin = A[i];

amax = INTMIN;
for(i=0; i<n; i++)
   if(A[i] > amax) amax = A[i];
```

```
amin = INTMIN;
amax = INTMAX;
for(i=0; i<n; i++) {
    int atmp = A[i];
     if(atmp < amin) amin = atmp;
     if(atmp > amax) amax = atmp;
}
```

© Saman Amarasinghe 2009

Eliminate Wasted Iterations

Change the loop bounds so that it will not iterate over an empty loop body

Viability?

- For a lot of iterations the loop body is empty
- Can change the loop bounds to make the loop tighter
- Or ability to change the data structures around (efficiently and correctly)

Example I

```
for(i=0; i<n; i++)
  for(j = i;  j < n – i; j++)
     …
```

```
for(i=0; i<n/2; i++)
  for(j = i; j < n – i; j++)
     …
```

© Saman Amarasinghe 2009

Example II

```
int val[N];
int visited[N];

for(i=0; i < N; i++)
    visited[i] = 0;

for(i=0;  i < N;  i++) {
    int minloc;
    int minval = MAXINT;
    for(j=0; j  < N;  j++)
        if(!visited[j])
            if(val[j] < minval) {
                minval = val[j];
                minloc = j;
            }
    visited[minloc] = 1;
     // process val[minloc]
}
```

```
int  val[N];

for(i=0;  i < N;  i++) {
    int minloc;
    int minval = MAXINT;
    for(j=0; j < N – i; j++)
        if(val[j] < minval) {
            minval = val[j];
            minloc = j ;
        }
     // process val[minloc]
     val[minloc] = val[N –  i – 1];
}
```

© Saman Amarasinghe 2009

Bentley's Rules

A. Modifying Data

B. Modifying Code

1. Loop Rules
2. Logic Rules
    a. Exploit Algebraic Identities
    b. Short Circuit Monotone functions
    c. Reordering tests
    d. Precompute Logic Functions
    e. Boolean Variable Elimination
3. Procedure Rules
4. Expression Rules
5. Parallelism Rules

© Saman Amarasinghe 2009

Exploit Algebraic Identities

If the evaluation of a logical expression is costly, replace algebraically equivalent

Examples

- sqr (x) > 0 →
- sqrt(x*x + y*y) < sqrt(a*a + b*b) →
- ln(A) + ln(B) →
- SIN(X)*SIN(X) + COS(X)*COS(X) →

© Saman Amarasinghe 2009

Short Circuit Monotone Functions

In checking if a monotonically increasing function is over a threshold, don't evaluate beyond the threshold

Example:

```
sum = 0;
for(i=0; i < N; i++)
   sum = sum + X[i];
if(sum > cutoff) return 0;
return 1;
```

```
sum = 0;
i = 0;
while((i < N) && (sum < cutoff))
   sum = sum + X[i++];
if(sum > cutoff) return 0;
return 1;
```

```
sum = 0;
i = 0;
X[N] = cutoff;
while (sum < cutoff)
   sum = sum + X[i++];
if(i == N) return 1;
return 0;
```

© Saman Amarasinghe 2009

Reordering tests

Logical tests should be arranged such that inexpensive and often successful tests precede expensive and rarely successful ones

Add inexpensive and often successful tests before expensive ones

Example:

```
if (sqrt(sqr(x1 – x2) +
          sqr(y1 – y2))  < rad1+rad2)
    return COLLITION;
return OK;
```

```
if(abs(x1-x2) > rad1+rad2)
  return OK;
if(abs(y1-y2) > rad1+rad2)
  return OK;
if (sqrt(sqr(x1 – x2) +
          sqr(y1 – y2))  < rad1+rad2)
    return COLLITION;
return OK;
```

© Saman Amarasinghe 2009

Precompute Logic Functions

A logical function over a small finite domain can be replaced by a lookup in a table that represents the domain.

Example:

```
int palindrome(unsigned char val)
{
  unsigned char l, r;
  int i;
  l = 0x01;
  r = 0x80;
  for(i=0; i <4; i++) {
    if(((val & l) == 0) ^ ((val & r) == 0))
      return 0;
    l = l << 1;
    r = r >> 1;
  }
  return 1;
}
```

```
int isPalindrome[256];

// initialize array

int palindrome(unsigned char val)
{
    return isPalindrome[val];
}
```

© Saman Amarasinghe 2009

Boolean Variable Elimination

Replace the assignment to a Boolean variable by re placing it by an IF-THEN-ELSE

Example:

```
int v;
v = Boolean expression;

S1;
if(v)
    S2;
else
    S3;
S4;
if(v)
    S5;
```

```
if(Boolean expression) {
    S1;
    S2;
    S4;
    S5;
}else {
    S1;
    S3;
    S4;
}
```

© Saman Amarasinghe 2009

Bentley's Rules

A. Modifying Data

B. Modifying Code

1. Loop Rules
2. Logic Rules
3. Procedure Rules
   a. Collapse Procedure Hierarchies
   b. Coroutines
   c. Tail Recursion Elimination
4. Expression Rules
5. Parallelism Rules

© Saman Amarasinghe 2009

Collapse Procedure Hierarchies

Inline small functions into the main body

- Eliminates the call overhead
- Provide further opportunities for compiler optimization

```
int getmax() {
    int xmax, i;
    xmax = MININT;
    for(i=0; i < N; i++)
        xmax = max(xmax, X[i]);
    return xmax;
}
```

```
#define max(a, b) (((a) > (b))?(a):(b))

int getmax() {
    int xmax, i;
    xmax = MININT;
    for(i=0; i < N; i++)
        xmax = max(xmax, X[i]);
    return xmax;
}
```

```
inline int max(int a,  int b) {
  if(a > b) return a;
  return b;
}

int getmax() {
    int xmax, i;
    xmax = MININT;
    for(i=0; i < N; i++)
        xmax = max(xmax, X[i]);
    return xmax;
}
```

© Saman Amarasinghe 2009

Coroutines

Multiple passes over data should be combined

- Similar to loop fusion(B.1.f), but at a scale and complexity beyond a single loop

Example pattern

```
Loop {
   Read from I
   ProcessA
   Write to II
}
Loop {
   Read from II
   ProcessB
   Write to III
}
```

```
Loop {
   Loop {
      Read from I
      ProcessA
      Write to buffer
   }
   Loop {
      Read from buffer
      Process B
      Write to III
   }
}
```

© Saman Amarasinghe 2009

Tail Recursion Elimination

In a self recursive function, if the last action is calling itself, eliminate the recursion.

Example pattern

```
int fact(int n,  int res) {
  if(n == 1) return res;
  return fact(n – 1,  res*n);
}
```

```
int fact(int n, int res) {
   while(1) {
      if(n == 1) return res;
      res = res*n;
      n = n  – 1;
   }
}
```

© Saman Amarasinghe 2009

Bentley's Rules

A. Modifying Data

B. Modifying Code

1. Loop Rules
2. Logic Rules
3. Procedure Rules
4. Expression Rules
    a. Compile-time Initialization
    b. Common Subexpression Elimination
    c. Pairing Computation
5. Parallelism Rules

© Saman Amarasinghe 2009

Compile-Time Initialization

If a value is a constant, make it a compile-time constant.

- Save the effort of calculation
- Allow value inlining
- More optimization opportunities

Example

```
……..
vol = 2 * pi() * r * r;
```

```
#define PI 3.14159265358979
#define  R  12

{
    ……..
    vol = 2 * PI * R * R;
```

© Saman Amarasinghe 2009

Common Subexpression Elimination

If the same expression is evaluated twice, do it only once

Viability?

- Expression has no side effects
- The expression value does not change between the evaluations
- The cost of keeping a copy is amortized by the complexity of the expression
- Too complicated for the compiler to do it automatically

Example

```
x = sin(a) * sin(a);
```

```
double tmp;
…
tmp = sin(a);
x = tmp * tmp;
```

© Saman Amarasinghe 2009

Pairing Computation

If two similar functions are called with the same arguments close to each other in many occasions, combine them.

- Reduce call overhead
- Possibility of sharing the computation cost
- More optimization possibilities

Example

```
x = r * cos(a);
y = r * sin(a);
```

```
typedef struct twodouble {
    double d1;
    double d2;
}
....
    twodouble dd;
     dd = sincos(a);
     x = r * dd.d1;
     y = r * dd.d2;
```

© Saman Amarasinghe 2009

Bentley's Rules

A. Modifying Data

B. Modifying Code

1. Loop Rules
2. Logic Rules
3. Procedure Rules
4. Expression Rules
5. Parallelism Rules
    a. Exploit Implicit Parallelism
    b. Exploit Inner Loop Parallelism
    c. Exploit Coarse Grain Parallelism
    d. Extra computation to create parallelism

© Saman Amarasinghe 2009

Implicit Parallelism

Reduce the loop carried dependences so that "software pipelining" can execute a compact schedule without stalls.

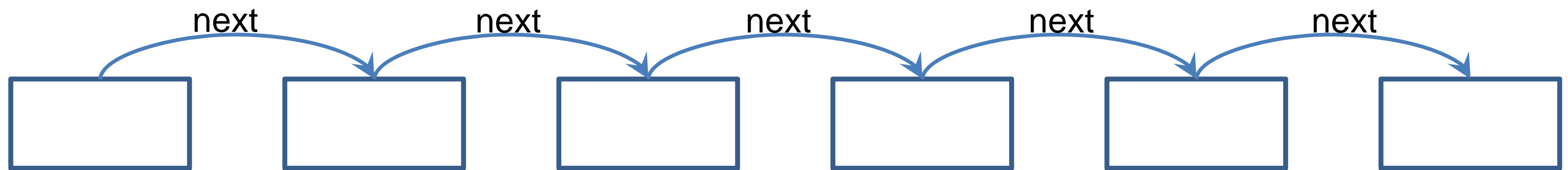
Example:

```
xmax = MININT;
for(i=0; i < N; i++)
   if(X[i] > xmax) xmax = X[i];
```

```
xmax1 = MININT;
xmax2 = MININT;
for(i=0; i < N –1; I += 2) {
   if(X[i] > xmax1) xmax1 = X[i];
   if(X[i+1] > xmax2) xmax2 = X[i+1];
}
if((i < N) &&(X[i] > xmax1)) xmax1 = X[i];
xmax = (xmax1 > xmax2)?xmax1:xmax2;
```

© Saman Amarasinghe 2009

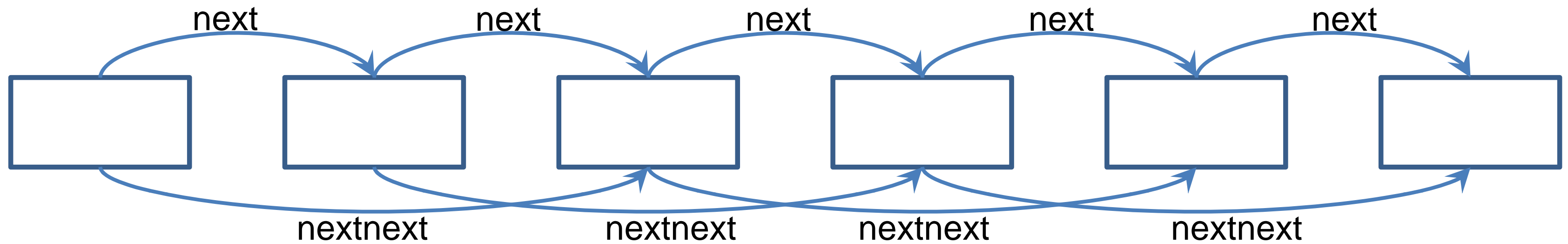
Example 2




```
curr = head;
tot = 0;
while(curr != NULL) {
    tot = tot + curr→val;
    curr = curr→next;
}
return tot;
```

© Saman Amarasinghe 2009

Example 2




```
curr = head;
tot = 0;
while(curr != NULL) {
    tot = tot + curr→val;
    curr = curr→next;
}
return tot;
```

```
curr = head;
if(curr == NULL) return 0;
tot1 = 0;
tot2 = 0;
while(curr→next) {
    tot1 = tot1 + curr→val;
    tot2 = tot2 + curr→next→val;
    curr = curr→nextnext;
}
if(curr)
    tot1 = tot1 + curr→val;
return tot1 + tot2;
```

Also see Rule A.1.a Data Structure Augmentation

© Saman Amarasinghe 2009

Exploit Inner Loop Parallelism

Facilitate inner loop vectorization (for SSE type instructions)

How? → by gingerly guiding the compiler to do so

- Iterative process by looking at why the loop is not vectorized and fixing those issues
- Most of the rules above can be used to simplify the loop so that the compiler can vectorize it

© Saman Amarasinghe 2009

Exploit Coarse Grain Parallelism

- Outer loop parallelism (doall and doacross loops)
- Task parallelism
- Ideal for multicores

- You need to do the parallelism yourself → later lectures

© Saman Amarasinghe 2009

Extra Computation to Create Parallelism

In many cases doing a little more work (or a slower algorithm) can make a sequential program a parallel one. Parallel execution may amortize the cost

Example:

```
double tot;
tot = 0;
for(i = 0; i < N; i++)
    for(j = 0;  j < N; j++)
        tot = tot + A[i][j];
```

```
double tot;
double tottmp[N];
for(i = 0; i < N; i++)
    tottmp[i] = 0;
for(i = 0; i < N; i++) {  //parallelizable
    double tmp;
    for(j = 0;  j < N; j++)
        tmp = tmp + A[i][j];
     tottmp[i]= tottmp[i]+ tmp;
}
tot = 0;
for(i = 0; i < N; i++)
    tot = tot + tottmp[i];
```

© Saman Amarasinghe 2009

Bentley's Rules

A Modifying Data

1. Space for Time
   a. Data Structure Augmentation
   b. Storing Precomputed Results
   c. Caching
   d. Lazy Evaluation
2. Time for Space
   a. Packing/Compression
   b. Interpreters
3. Space and Time
   a. SIMD

B Modifying Code

1. Loop Rules
   a. Loop Invariant Code Motion
   b. Sentinel Loop Exit Test
   c. Loop Elimination by Unrolling
   d. Partial Loop Unrolling
   e. Loop fusion
   f. Eliminate wasted iterations
2. Logic Rules
   a. Exploit Algebraic Identities
   b. Short Circuit Monotone functions
   c. Reordering tests
   d. Precompute Logic Functions
   e. Boolean Variable Elimination
3. Procedure Rules
   a. Collapse Procedure Hierarchies
   b. Coroutines
   c. Tail Recursion Elimination
4. Expression Rules
   a. Compile-time Initialization
   b. Common Subexpression Elimination
   c. Pairing Computation
5. Parallelism Rules
   a. Exploit Implicit Parallelism
   b. Exploit Inner Loop Parallelism
   c. Exploit Coarse Grain Parallelism
   d. Extra computation to create parallelism

© Saman Amarasinghe 2009

Traveling Salesman Problem

Definition

- List of cities
- Location of each city (x,y coordinates on a 2-D map)
- Need to visit all the cities
- What order to visit the cities so that the distance traveled is shortest

Exact Shortest Distance Algorithm → Exponential

A Good Greedy Heuristic

- Start with any city
- Find the closest city that haven't been visited
- Visit that city next
- Iterate until all the cities are visited

© Saman Amarasinghe 2009

TSP Example

Original

```
  void get_path_1(int path[]) {
    int visited[MAXCITIES];
    int i, j;
    int curr, closest;
    double cdist;
    double totdist;

    for(i=0; i <MAXCITIES; i++)
      visited[i] = 0;

  curr = MAXCITIES-1;
    visited[curr] = 1;
    totdist = 0;
    path[0] = curr;
    for(i=1; i < MAXCITIES; i++) {
      cdist = MAXDOUBLE;
      for(j=0; j < MAXCITIES; j++)
        if(visited[j] == 0)
          if(dist(curr, j) < cdist) {
            cdist = dist(curr, j);
            closest = j;
          }

      path[i] = closest;
      visited[closest] = 1;
      totdist += dist(curr, closest);
      curr = closest;
    }
}

double dist(int i, int j) {
  return sqrt((Cities[i].x - Cities[j].x)*(Cities[i].x - Cities[j].x)+
              (Cities[i].y - Cities[j].y)*(Cities[i].y - Cities[j].y));
}
```

© Saman Amarasinghe 2009

TSP Example

Original

```
void get_path_1(int path[]) {
  int visited[MAXCITIES];
  int i, j;
  int curr, closest;
  double cdist;
  double totdist;

  for(i=0; i <MAXCITIES; i++)
    visited[i] = 0;

 curr = MAXCITIES-1;
  visited[curr] = 1;
  totdist = 0;
  path[0] = curr;
  for(i=1; i < MAXCITIES; i++) {
    cdist = MAXDOUBLE;
    for(j=0; j < MAXCITIES; j++)
      if(visited[j] == 0)
        if(dist(curr, j) < cdist) {
          cdist = dist(curr, j);
          closest = j;
        }

    path[i] = closest;
    visited[closest] = 1;
    totdist += dist(curr, closest);
    curr = closest;
  }
}

double dist(int i, int j) {
  return sqrt((Cities[i].x - Cities[j].x)*(Cities[i].x - Cities[j].x)+
              (Cities[i].y - Cities[j].y)*(Cities[i].y - Cities[j].y));
}
```

B.2.a Exploit Algebraic Identities

```
void get_path_2(int path[]) {
  int visited[MAXCITIES];
  int i, j;
  int curr, closest;
  double cdist;
  double totdist;

  for(i=0; i <MAXCITIES; i++)
    visited[i] = 0;

  curr = MAXCITIES-1;
  visited[curr] = 1;
  totdist = 0;
  path[0] = curr;
  for(i=1; i < MAXCITIES; i++) {
    cdist = MAXDOUBLE;
    for(j=0; j < MAXCITIES; j++)
      if(visited[j] == 0)
        if(distsq(curr, j) < cdist) {
          cdist = distsq(curr, j);
          closest = j;
        }

    path[i] = closest;
    visited[closest] = 1;
    totdist += dist(curr, closest);
    curr = closest;
  }
}

double distsq(int i, int j) {
  return ((Cities[i].x - Cities[j].x)*(Cities[i].x - Cities[j].x)+
          (Cities[i].y - Cities[j].y)*(Cities[i].y - Cities[j].y));
}
```

© Saman Amarasinghe 2009

TSP Example

B.2.a Exploit Algebraic Identities

```
void get_path_2(int path[])
{
  int visited[MAXCITIES];
  int i, j;
  int curr, closest;
  double cdist;
  double totdist;

  for(i=0; i <MAXCITIES; i++)
    visited[i] = 0;

  curr = MAXCITIES-1;
  visited[curr] = 1;
  totdist = 0;
  path[0] = curr;
  for(i=1; i < MAXCITIES; i++) {
    cdist = MAXDOUBLE;
    for(j=0; j < MAXCITIES; j++)
      if(visited[j] == 0)

        if(distsq(curr, j) < cdist) {
          cdist = distsq(curr, j);
          closest = j;
        }


    path[i] = closest;
    visited[closest] = 1;
    totdist += dist(curr, closest);
    curr = closest;
  }
}

double distsq(int i, int j) {
  return ((Cities[i].x - Cities[j].x)*(Cities[i].x - Cities[j].x)+
              (Cities[i].y - Cities[j].y)*(Cities[i].y - Cities[j].y));
}
```

B.4.b Common Subexpression Elimination

```
void get_path_3(int path[])
{
  int visited[MAXCITIES];
  int i, j;
  int curr, closest;
  double cdist, tdist;
  double totdist;

  for(i=0; i <MAXCITIES; i++)
    visited[i] = 0;

  curr = MAXCITIES-1;
  visited[curr] = 1;
  totdist = 0;
  path[0] = curr;
  for(i=1; i < MAXCITIES; i++) {
    cdist = MAXDOUBLE;
    for(j=0; j < MAXCITIES; j++)
      if(visited[j] == 0) {
        tdist = distsq(curr, j);
        if(tdist < cdist) {
          cdist = tdist;
          closest = j;
        }
      }

    path[i] = closest;
    visited[closest] = 1;
    totdist += dist(curr, closest);
    curr = closest;
  }
}
```

© Saman Amarasinghe 2009

TSP Example

B.4.b Common Subexpression Elimination

```
void get_path_3(int path[])
{
  int visited[MAXCITIES];
  int i, j;
  int curr, closest;
  double cdist, tdist;
  double totdist;

  for(i=0; i <MAXCITIES; i++)
    visited[i] = 0;

  curr = MAXCITIES-1;
  visited[curr] = 1;
  totdist = 0;
  path[0] = curr;
  for(i=1; i < MAXCITIES; i++) {
    cdist = MAXDOUBLE;


    for(j=0; j < MAXCITIES; j++)
      if(visited[j] == 0) {
        tdist = distsq(curr, j);

        if(tdist < cdist) {
          cdist = tdist;
          closest = j;
        }
      }

    path[i] = closest;
    visited[closest] = 1;
    totdist += dist(curr, closest);
    curr = closest;
  }
}
```

B.1.a Loop Invariant Code Motion

```
void get_path_4(int path[])
{
  int visited[MAXCITIES];
  int i, j;
  int curr, closest;
  double cdist, tdist;
  double cx, cy;
  double totdist;

  for(i=0; i <MAXCITIES; i++)
    visited[i] = 0;

curr = MAXCITIES-1;
  visited[curr] = 1;
  totdist = 0;
  path[0] = curr;
  for(i=1; i < MAXCITIES; i++) {
    cdist = MAXDOUBLE;
    cx = Cities[curr].x;
    cy = Cities[curr].y;
    for(j=0; j < MAXCITIES; j++)
      if(visited[j] == 0) {
        tdist = (Cities[j].x - cx)*(Cities[j].x - cx) +
                (Cities[j].y - cy)*(Cities[j].y - cy);
        if(tdist < cdist) {
          cdist = tdist;
          closest = j;
        }
      }

    path[i] = closest;
    visited[closest] = 1;
    totdist += dist(curr, closest);
    curr = closest;
  }
}
```

© Saman Amarasinghe 2009

TSP Example

B.1.a Loop Invariant Code Motion

```
void get_path_4(int path[])
{
  int visited[MAXCITIES];
  int i, j, curr, closest;
  double cdist, tdist, cx, cy, totdist;

  for(i=0; i <MAXCITIES; i++)
    visited[i] = 0;

  curr = MAXCITIES-1;
  visited[curr] = 1;
  totdist = 0;
  path[0] = curr;
  for(i=1; i < MAXCITIES; i++) {
    cdist = MAXDOUBLE;
    cx = Cities[curr].x;
    cy = Cities[curr].y;
    for(j=0; j < MAXCITIES; j++)
      if(visited[j] == 0) {
        tdist = (Cities[j].x - cx)*(Cities[j].x - cx) +
                (Cities[j].y - cy)*(Cities[j].y - cy);

        if(tdist < cdist) {
          cdist = tdist;
          closest = j;
        }
      }

    path[i] = closest;
    visited[closest] = 1;
    totdist += dist(curr, closest);
    curr = closest;
  }
}
```

B.1.c Reordering Tests

```
void get_path_5(int path[])
{
  int visited[MAXCITIES];
  int i, j, curr, closest;
  double cdist, tdist, cx, cy, totdist;

  for(i=0; i <MAXCITIES; i++)
    visited[i] = 0;

  curr = MAXCITIES-1;
  visited[curr] = 1;
  totdist = 0;
  path[0] = curr;
  for(i=1; i < MAXCITIES; i++) {
    cdist = MAXDOUBLE;
    cx = Cities[curr].x;
    cy = Cities[curr].y;
    for(j=0; j < MAXCITIES; j++)
      if(visited[j] == 0) {
        tdist = (Cities[j].x - cx)*(Cities[j].x - cx);
        if(tdist < cdist) {
          tdist += (Cities[j].y - cy)*(Cities[j].y - cy);
          if(tdist < cdist) {
            cdist = tdist;
            closest = j;
          }
        }
      }
    path[i] = closest;
    visited[closest] = 1;
    totdist += dist(curr, closest);
    curr = closest;
  }
}
```

© Saman Amarasinghe 2009

TSP Example

B.1.c Reordering Tests

```
void get_path_5(int path[])
{
  int visited[MAXCITIES];
  int i, j, curr, closest;
  double cdist, tdist, cx, cy, totdist;

  for(i=0; i <MAXCITIES; i++)
    visited[i] = 0;

  curr = MAXCITIES-1;
  visited[curr] = 1;
  totdist = 0;
  path[0] = curr;
  for(i=1; i < MAXCITIES; i++) {
    cdist = MAXDOUBLE;
    cx = Cities[curr].x;
    cy = Cities[curr].y;
    for(j=0; j < MAXCITIES; j++)
      if(visited[j] == 0) {
        tdist = (Cities[j].x - cx)*(Cities[j].x - cx);

        if(tdist < cdist) {
          tdist += (Cities[j].y - cy)*(Cities[j].y - cy);

          if(tdist < cdist) {
            cdist = tdist;
            closest = j;
          }
        }
      }
    path[i] = closest;
    visited[closest] = 1;
    totdist += dist(curr, closest);
    curr = closest;
  }
}
```

B.4.b Common Subexpression Elimination

```
void get_path_6(int path[])
{
  int visited[MAXCITIES];
  int i, j, curr, closest;
  double cdist, tdist, cx, cy, totdist;

  for(i=0; i <MAXCITIES; i++)
    visited[i] = 0;

  curr = MAXCITIES-1;
  visited[curr] = 1;
  totdist = 0;
  path[0] = curr;
  for(i=1; i < MAXCITIES; i++) {
    cdist = MAXDOUBLE;
    cx = Cities[curr].x;
    cy = Cities[curr].y;
    for(j=0; j < MAXCITIES; j++)
      if(visited[j] == 0) {
        double tx =(Cities[j].x - cx);
        tdist = tx*tx;
        if(tdist < cdist) {
          double ty = (Cities[j].y - cy);
          tdist += ty*ty;
          if(tdist < cdist) {
            cdist = tdist;
            closest = j;
          }
        }
      }
    path[i] = closest;
    visited[closest] = 1;
    totdist += dist(curr, closest);
    curr = closest;
  }
}
```

© Saman Amarasinghe 2009

TSP Example

B.4.b Common Subexpression Elimination

```
void get_path_6(int path[])
{
  int visited[MAXCITIES];
  int i, j, curr, closest;
  double cdist, tdist, cx, cy, totdist;
  for(i=0; i <MAXCITIES; i++)
    visited[i] = 0;

  curr = MAXCITIES-1;
  visited[curr] = 1;
  totdist = 0;
  path[0] = curr;
  for(i=1; i < MAXCITIES; i++) {
    cdist = MAXDOUBLE;
    cx = Cities[curr].x;
    cy = Cities[curr].y;
    for(j=0; j < MAXCITIES; j++)
      if(visited[j] == 0) {

        double tx =(Cities[j].x - cx);
        tdist = tx*tx;
        if(tdist < cdist) {
          double ty = (Cities[j].y - cy);
          tdist += ty*ty;
          if(tdist < cdist) {
            cdist = tdist;
            closest = j;

          }
        }
      }
    path[i] = closest;
    visited[closest] = 1;
    totdist += dist(curr, closest);
    curr = closest;
  }
}
```

B.1.f Eliminate Wasted Iterations

```
void get_path_7(int path[])
{
  int unvisited[MAXCITIES];
  int i, j, k, curr, closek, closej;
  double cdist, tdist, cx, cy, totdist;
  for(i=0; i <MAXCITIES; i++)
    unvisited[i] = i;

  curr = MAXCITIES-1;

  totdist = 0;
  path[0] = curr;
  for(i=1; i < MAXCITIES; i++) {
    cdist = MAXDOUBLE;
    cx = Cities[curr].x;
    cy = Cities[curr].y;
    for(j=0; j < MAXCITIES-i; j++) {
      double tx;
      k = unvisited[j];
      tx =(Cities[k].x - cx);
      tdist = tx*tx;
      if(tdist < cdist) {
        double ty = (Cities[k].y - cy);
        tdist += ty*ty;
        if(tdist < cdist) {
          cdist = tdist;
          closek = k;
          closej = j;
        }
      }
    }
    path[i] = closek;
    unvisited[closej] = unvisited[MAXCITIES - i - 1];
    totdist += dist(curr, closek);
    curr = closek;
  }
}
```

© Saman Amarasinghe 2009

TSP Example

B.4.b Common Subexpression Elimination

```
void get_path_6(int path[])
{
 int visited[MAXCITIES];
 int i, j, curr, closest;
 double cdist, tdist, cx, cy, totdist;
 for(i=0; i <MAXCITIES; i++)
  visited[i] = 0;
 curr = MAXCITIES-1;
 visited[curr] = 1;
 totdist = 0;
 path[0] = curr;
 for(i=1; i < MAXCITIES; i++) {
  cdist = MAXDOUBLE;
  cx = Cities[curr].x;
  cy = Cities[curr].y;
  for(j=0; j < MAXCITIES; j++)
   if(visited[j] == 0) {
     double tx =(Cities[j].x - cx);
     tdist = tx*tx;
     if(tdist < cdist) {
      double ty = (Cities[j].y - cy);
      tdist += ty*ty;
      if(tdist < cdist) {
       cdist = tdist;
       closest = j;
      }
     }
   }
  path[i] = closest;
  visited[closest] = 1;
  totdist += dist(curr, closest);

  curr = closest;
 }
}
```

B.1.f Eliminate Wasted Iterations

```
void get_path_8(int path[])
{

 int i, j, curr, closest;
 double cdist, tdist, cx, cy, totdist;


 curr = MAXCITIES-1;

 totdist = 0;
 path[0] = curr;
 for(i=1; i < MAXCITIES; i++) {
  cdist = MAXDOUBLE;
  cx = Cities[curr].x;
  cy = Cities[curr].y;
  for(j=0; j < MAXCITIES-i; j++)
   if(curr != j) {
     double tx =(Cities[j].x - cx);
     tdist = tx*tx;
     if(tdist < cdist) {
      double ty = (Cities[j].y - cy);
      tdist += ty*ty;
      if(tdist < cdist) {
       cdist = tdist;
       closest = j;
      }
     }
   }
  path[i] = closest;

  totdist += dist(curr, closest);
  Cities[curr] = Cities[MAXCITIES - i];
  curr = closest;
 }
}
```

© Saman Amarasinghe 2009

Performance

			Result		Time				
1		Original	96.344605		519.0697	ms	1.52		
2	B.2.a	Exploit Algebric Identities	96.344605		341.0185	ms		1.08	1.52
3	B.4.b	Common Subexpression Elimination	96.344605		314.5534	ms	1.02		1.65
4	B.1.a	Loop Invariant code Motion	96.344605		308.9298	ms		1.02	1.68
5	B.1.c	Reordering Tests	96.344605		302.6724	ms	0.91		1.71
6	B.4.b	Common Subexpression Elimination	96.344605		331.6027	ms		2.7	1.57
7	B.1.f	Eliminate Wated Iterations	97.738769		123.7693	ms	1.23		4.19
8	B.1.f	Eliminate Wated Iterations	99.003897		100.6096	ms			5.16

© Saman Amarasinghe 2009

MIT OpenCourseWare
http://ocw.mit.edu

6.172 Performance Engineering of Software Systems
Fall 2010

For information about citing these materials or our Terms of Use, visit: http://ocw.mit.edu/terms.