CSCI-580 Advanced High Performance Computing

Performance Hacking: Rules and Techniques for Optimization

Bo Wu
Colorado School of Mines

Slide resource from: Saman Amarasinghe (MIT)
Definition.
The work of a program (on a given input) is the sum total of all the operations executed by the program.
Optimizing work

- Algorithm design can produce dramatic reductions in the amount of work it takes to solve a problem, as when a $\Theta(n \lg n)$-time sort replaces a $\Theta(n^2)$-time sort.

- Does reducing work always improve performance?
  - No. The hardware is complicated.
    - instruction-level parallelism
    - caching
    - vectorization

- Nevertheless, reducing the work serves as a good heuristic for reducing overall running time.
How to reduce work

- 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 set of rules for reducing work
  - Patterns that occur regularly
  - Mistakes many make
Optimizing Compilers are also tasked with reducing the work.
   – If you are lucky the compiler will do all the work reduction for you
   – Coaxing a reluctant compiler is easier than doing it yourself
   – You should only get involved when compiler is unable to do

You don’t want to interfere and make the compiler unable to do what it can
   – An example?
Jon Louis Bentley

Writing Efficient Programs

1982
Before we talk about the techniques, what’s the first thing to do before optimizations?
The idea of **packing** is to store more than one data value in a machine word. The related idea of **encoding** is to convert data values into a representation requiring fewer bits.
Packing and encoding

The idea of **packing** is to store more than one data value in a machine word. The related idea of **encoding** is to convert data values into a representation requiring fewer bits.

**Example: Encoding dates**

- The string “February 14, 2008” can be stored in 19 bytes (null terminating byte included), which means that 3 double (64-bit) words must moved whenever a date is manipulated using this representation.
Packing and encoding

The idea of **packing** is to store more than one data value in a machine word. The related idea of **encoding** is to convert data values into a representation requiring fewer bits.

**Example:** Encoding dates

- The string “February 14, 2008” can be stored in 19 bytes (null terminating byte included), which means that 3 double (64-bit) words must moved whenever a date is manipulated using this representation.

- Assuming that we only store years between 1 C.E. and 4096 C.E., there are about $365.25 \times 4096 \approx 1.5$ M dates, which can be encoded in $\lceil \log(1.5 \times 10^6) \rceil = 21$ bits, which fits in a single (32-bit) word.
Packing and encoding

The idea of packing is to store more than one data value in a machine word. The related idea of encoding is to convert data values into a representation requiring fewer bits.

Example: Encoding dates

- The string “February 14, 2008” can be stored in 19 bytes (null terminating byte included), which means that 3 double (64-bit) words must moved whenever a date is manipulated using this representation.

- Assuming that we only store years between 1 C.E. and 4096 C.E., there are about $365.25 \times 4096 \approx 1.5$ M dates, which can be encoded in $\lceil \lg(1.5 \times 10^6) \rceil = 21$ bits, which fits in a single (32-bit) word.

- But querying the month of a date takes more work.
Packing and encoding

Example: Packing dates

- Instead, let us pack the three fields into a word:

```c
typedef struct {
    unsigned int year: 12;
    unsigned int month: 4;
    unsigned int day: 5;
} date_t;
```

- This packed representation still only takes 21 bits, but the individual fields can be extracted much more quickly than if we had encoded the 1.5 M dates as sequential integers.
Packing and encoding

Example: Packing dates

- Instead, let us pack the three fields into a word:

```c
typedef struct {
    unsigned int year: 12;
    unsigned int month: 4;
    unsigned int day: 5;
} date_t;
```

- This packed representation still only takes 21 bits, but the individual fields can be extracted much more quickly than if we had encoded the 1.5 M dates as sequential integers.

Sometimes unpacking and decoding are the optimization, depending on whether more work is involved moving the data or operating on it.
Augmentation
Augmentation
Augmentation

The idea of data-structure **augmentation** is to add information to a data structure to make common operations do less work.

**Example:** Appending singly linked lists

- Appending one list to another requires walking the length of the first list to set its null pointer to the start of the second.

- **Augmenting** the list with a tail pointer allows appending to operate in constant time.
The idea of **precomputation** is to perform calculations in advance so as to avoid doing them at “mission-critical” times.
The idea of **precomputation** is to perform calculations in advance so as to avoid doing them at “mission-critical” times.

**Example:** Binomial coefficients

\[
\binom{a}{b} = \frac{a!}{b!(a-b)!}
\]

Expensive to compute (lots of multiplications), and watch out for integer overflow for even modest values of \(a\) and \(b\).
Precomputation

The idea of **precomputation** is to perform calculations in advance so as to avoid doing them at “mission-critical” times.

**Example:** Binomial coefficients

\[
\binom{a}{b} = \frac{a!}{b!(a-b)!}
\]

Expensive to compute (lots of multiplications), and watch out for integer overflow for even modest values of \(a\) and \(b\).

**Idea:** Precompute the table of coefficients when initializing, and do table look-up at runtime.
The idea of **caching** is to store results that have been accessed recently so that the program need not compute them again.

```cpp
inline double hypotenuse(double A, double B) {
    return sqrt(A * A + B * B);
}

double cached_A = 0.0;
double cached_B = 0.0;
double cached_h = 0.0;

inline double hypotenuse(double A, double B) {
    if (A == cached_A && B == cached_B) {
        return cached_h;
    }
    cached_A = A;
    cached_B = B;
    cached_h = sqrt(A * A + B * B);
    return cached_h;
}
```
Caching

The idea of **caching** is to store results that have been accessed recently so that the program need not compute them again.

```
inline double hypotenuse(double A, double B) {
    return sqrt(A * A + B * B);
}
```

About **30%** faster if cache is hit **2/3** of the time.

```
double cached_A = 0.0;
double cached_B = 0.0;
double cached_h = 0.0;

inline double hypotenuse(double A, double B) {
    if (A == cached_A && B == cached_B) {
        return cached_h;
    }
    cached_A = A;
cached_B = B;
cached_h = sqrt(A * A + B * B);
    return cached_h;
}
```
Lazy evaluation

- Delay the evaluation of expressions until they are used
Lazy evaluation

- Delay the evaluation of expressions until they are used

In Python 2.X

```python
>>> r = range(10)
>>> print r
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> print r[3]
3
```
Lazy evaluation

- Delay the evaluation of expressions until they are used

In Python 2.X

```python
>>> r = range(10)
>>> print r
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> print r[3]
3
```

In Python 3.X

```python
>>> r = range(10)
>>> print(r)
range(0, 10)
>>> print(r[3])
3
```
Sparsity

The idea of exploiting sparsity is to avoid storing and computing on zeroes. “The fastest way to compute is not to compute at all.”
Sparsity

The idea of exploiting sparsity is to avoid storing and computing on zeroes. “The fastest way to compute is not to compute at all.”

Example: Sparse matrix multiplication

\[
A = \begin{pmatrix}
3 & 0 & 0 & 0 & 1 & 0 \\
0 & 4 & 1 & 0 & 5 & 9 \\
0 & 0 & 0 & 2 & 0 & 6 \\
5 & 0 & 0 & 3 & 0 & 0 \\
0 & 0 & 0 & 0 & 5 & 0 \\
0 & 0 & 0 & 8 & 9 & 7 \\
\end{pmatrix}
\]

\[
\times \begin{pmatrix}
1 \\
2 \\
3 \\
4 \\
5 \\
6 \\
\end{pmatrix}
\]

Dense matrix–vector multiplication performs \( n^2 = 36 \) scalar multiplies, but only 14 entries are nonzero.
Sparsity (2)

\[
\begin{pmatrix}
3 & 0 & 0 & 0 & 1 & 0 \\
0 & 4 & 1 & 0 & 5 & 9 \\
0 & 0 & 0 & 2 & 0 & 6 \\
5 & 0 & 0 & 3 & 0 & 0 \\
0 & 0 & 0 & 0 & 5 & 0 \\
0 & 0 & 0 & 8 & 9 & 7
\end{pmatrix}
\]

**Compressed Sparse Rows (CSR)**

- rows: 0 2 6 8 10 11 14
- cols: 0 4 1 2 4 5 3 5 0 3 0 4 3 4
- vals: 3 1 4 1 5 9 2 6 5 3 5 8 9 7
Sparsity (3)

CSR matrix-vector multiplication

typedef struct {
    int n, nnz;
    int *rows;    // length n
    int *cols;    // length nnz
    double *vals; // length nnz
} sparse_matrix_t;

void spmv(sparse_matrix_t *A, double *x, double *y) {
    for (int i = 0; i < A->n; i++) {
        y[i] = 0;
        for (int k = A->rows[i]; k < A->rows[i+1]; k++) {
            int j = A->cols[k];
            y[i] += A->vals[k] * x[j];
        }
    }
}

Number of scalar multiplications = nnz, which is potentially much less than $n^2$. 
When performing a series of tests, the idea of **short-circuiting** is to stop evaluating as soon as you know the answer.

```c
#include <stdbool.h>

bool sum_exceeds(int *A, int n, int limit) {
    int sum = 0;
    for (int i = 0; i < n; i++) {
        sum += A[i];
    }
    return sum > limit;
}
```

```c
#include <stdbool.h>

bool sum_exceeds(int *A, int n, int limit) {
    int sum = 0;
    for (int i = 0; i < n; i++) {
        sum += A[i];
        if (sum > limit) {
            return true;
        }
    }
    return false;
}
```
Consider code that executes a sequence of logical tests. The idea of **ordering tests** is to perform those that are more often “successful” — a particular alternative is selected by the test — before tests that are rarely successful. Similarly, inexpensive tests should precede expensive ones.
Consider code that executes a sequence of logical tests. The idea of \texttt{ordering tests} is to perform those that are more often “successful” — a particular alternative is selected by the test — before tests that are rarely successful. Similarly, inexpensive tests should precede expensive ones.

```c
#include <stdbool.h>

bool is_whitespace(char c) {
    if (c == ' ' || c == '
' || c == '	' || c == '') {
        return true;
    }
    return false;
}
```

```c
#include <stdbool.h>

bool is_whitespace(char c) {
    if (c == ' ' || c == '
' || c == '	' || c == '') {
        return true;
    }
    return false;
}
```
The idea of **combining tests** is to replace a sequence of tests with one test or switch.
Combining Tests

The idea of combining tests is to replace a sequence of tests with one test or switch.

Full adder

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>carry</th>
<th>sum</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

```c
void full_add (int a, int b, int c, int *sum, int *carry) {
    if (a == 0) {
        if (b == 0) {
            if (c == 0) {
                *sum = 0;
                *carry = 0;
            } else {
                *sum = 1;
                *carry = 0;
            }
        } else {
            *sum = 0;
            *carry = 1;
        }
    } else {
        if (c == 0) {
            *sum = 1;
            *carry = 0;
        } else {
            *sum = 0;
            *carry = 1;
        }
    }
}
```
The idea of **combining tests** is to replace a sequence of tests with one test or switch.

For this example, table look-up is even better!
Tail-recursion elimination

The idea of tail-recursion elimination is to replace a recursive call that occurs as the last step of a function with a branch, saving function-call overhead.
The idea of tail-recursion elimination is to replace a recursive call that occurs as the last step of a function with a branch, saving function-call overhead.

```c
void quicksort(int *A, int n) {
  if (n > 1) {
    int r = partition(A, n);
    quicksort(A, r);
    quicksort(A + r, n - r - 1);
  }
}
```

The idea of coarsening recursion is to increase the size of the base case and handle it with more efficient code that avoids function-call overhead.

```c
#define THRESHOLD 10
void quicksort(int *A, int n) {
    while (n > THRESHOLD) {
        int r = partition(A, n);
        quicksort (A, r);
        A += r + 1;
        n -= r + 1;
    }
    // insertion sort for small arrays
    for (int j = 1; j < n; ++j) {
        int key = A[j];
        int i = j - 1;
        while (i >= 0 && A[i] > key) {
            A[i+1] = A[i];
            --i;
        }
        A[i+1] = key;
    }
}
```
Strength reduction

- Idea: replace expensive operations with equivalent but less expensive operations

- Example:
  ```
  for(i=0; i<100; i++)
  t = a * i;
  t = 0;
  for(i=0; i<100; i++)
  t = t + a;
  ```

- Pros:
  - less expensive operations

- Cons:
  - more values to keep
  - introduce loop-carried dependence
Strength reduction: an example

```c
int sumcalc(int a, int b, int N)
{
    int i;
    int x, t, u;
    x = 0;
    u = (4*a/b);
    for(i = 0; i <= N; i++) {
        t = i+1;
        x = x + u*i + t*t;
    }
    return x;
}
```
**Strength reduction: an example**

```c
int sumcalc(int a, int b, int N)
{
    int i;
    int x, t, u;
    x = 0;
    u = (4*a/b);
    for(i = 0; i <= N; i++) {
        t = i+1;
        x = x + u*i + t*t;
    }
    return x;
}
```
Strength reduction: an example

```c
int sumcalc(int a, int b, int N)
{
    int i;
    int x, t, u, v;
    x = 0;
    u = (4*a/b);
    v = 0;
    for(i = 0; i <= N; i++) {
        t = i+1;
        x = x + u*i + t*t;
        v = v + u;
    }
    return x;
}
```
Swap

Problem
Swap two integers $x$ and $y$.

t = x;
x = y;
y = t;
Non-temp swap

Problem
Swap two integers $x$ and $y$ without using a temporary.

\[
\begin{align*}
x &= x \land y; \\
y &= x \land y; \\
x &= x \land y;
\end{align*}
\]

Example
\[
\begin{array}{cccc}
x & 10111101 & 10010011 & 10010011 & 00101110 \\
y & 00101110 & 00101110 & 10111101 & 10111101
\end{array}
\]
Non-temp swap

Problem
Swap two integers $x$ and $y$ without using a temporary.

Example

<table>
<thead>
<tr>
<th>$x$</th>
<th>10111101</th>
<th>10010011</th>
<th>10010011</th>
<th>00101110</th>
</tr>
</thead>
<tbody>
<tr>
<td>$y$</td>
<td>00101110</td>
<td>00101110</td>
<td>10111101</td>
<td>10111101</td>
</tr>
</tbody>
</table>
Non-temp swap

Problem
Swap two integers $x$ and $y$ without using a temporary.

$x = x \land y$;
$y = x \land y$;
$x = x \land y$;

Example

<table>
<thead>
<tr>
<th>x</th>
<th>10111101</th>
<th>10010011</th>
<th>10010011</th>
<th>00101110</th>
</tr>
</thead>
<tbody>
<tr>
<td>y</td>
<td>00101110</td>
<td>00101110</td>
<td><strong>10111101</strong></td>
<td>10111101</td>
</tr>
</tbody>
</table>
Non-temp swap

Problem
Swap two integers \( x \) and \( y \) without using a temporary.

\[
\begin{align*}
  x &= x \land y; \\
  y &= x \land y; \\
  x &= x \land y;
\end{align*}
\]

Example

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>( x )</td>
<td>10111101</td>
<td>10010011</td>
<td>10010011</td>
<td>00101110</td>
</tr>
<tr>
<td>( y )</td>
<td>00101110</td>
<td>00101110</td>
<td>10111101</td>
<td>10111101</td>
</tr>
</tbody>
</table>
Non-temp swap

**Problem**
Swap two integers $x$ and $y$ without using a temporary.

\[
x = x \oplus y;
\]
\[
y = x \oplus y;
\]
\[
x = x \oplus y;
\]

**Example**

<table>
<thead>
<tr>
<th></th>
<th>10111101</th>
<th>10010011</th>
<th>10010011</th>
<th>00101110</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x$</td>
<td>00101110</td>
<td>00101110</td>
<td>10111101</td>
<td>10111101</td>
</tr>
<tr>
<td>$y$</td>
<td>00101110</td>
<td>00101110</td>
<td>10111101</td>
<td>10111101</td>
</tr>
</tbody>
</table>

**Why it works**
XOR is its own inverse: $(x \oplus y) \oplus y = x$. 
Minimum of two integers

**Problem**
Find the minimum \( r \) of two integers \( x \) and \( y \).

```c
if (x < y)
    r = x;
else
    r = y;
```

or

```c
r = (x < y) ? x : y;
```
Minimum of two integers

Problem
Find the minimum $z$ of two integers $x$ and $y$ without a branch.

$$r = y \land ((x \land y) \land \neg(x < y));$$

Why it works:
• C represents the Booleans TRUE and FALSE with the integers 1 and 0, respectively.
• If $x < y$, then $-(x < y) = -1$, which is all 1’s in two’s complement representation. Therefore, we have $y \land (x \land y) = x$.
• If $x \geq y$, then $-(x < y) = 0$. Therefore, we have $y \land 0 = y$. 
Autotuning

- Idea: automatically find the optimal (near-optimal) parameters
- Tune the algorithm

- Tune the optimization parameter
  - Tiling size, prefetching aggressiveness
Autotuning

- Idea: automatically find the optimal (near-optimal) parameters
- Tune the algorithm

No single algorithm or optimization parameter can be the best for all programs and inputs

- Tune the optimization parameter
  - Tiling size, prefetching aggressiveness
Autotune a program

1. Get a candidate value
2. Compile the program
3. Calculate Ave. execution time
Autotune a single parameter

Depends on the values!
For the tile size
- Can use hill climbing

Hill Climbing
- Pick a value at random
- Find the positive gradient and move that way
- Until top of the hill found
- Can do binary search
Autotune a single parameter

- Depends on the values!
- For the tile size
  - Can use hill climbing

**Hill Climbing**
- Pick a value at random
- Find the positive gradient and move that way
- Until top of the hill found
- Can do binary search

**Stochastic Search**
Autotune compiler options
You’ve learned so many optimization techniques, I guess you want to measure the performance.
Ways to measure a program

- External measurements — time.
- Instrument the program — Change the program to measure it. E.g., call gettimeofday().
- Hardware and Operating Systems Support — run the program with counters maintained by the hardware and operating system. E.g., use perf.
Time

○ The time command measures the elapsed time, user time, and system time

○ Good for the whole program, but sometimes you want to measure part of the program
The “perf stat” command can measure hardware events, such as the number of page faults, cache misses or mispredicted branches.

Again, sometimes you want to measure part of a program.
function of the day()

To measure just part of your program:

```c
#include <sys/time.h>
...

// timeval has has two fields: tv_sec and tv_usec, 
// which are seconds and microseconds, respectively, 
// since the Epoch (1/1/1970).
struct timeval start, end;
gettimeofday(&start, NULL);
function_to_measure();
gettimeofday(&end, NULL);
double tdiff = (end.tv_sec-start.tv_sec) 
            + 1e-6*(end.tv_usec-start.tv_usec);
```

Note: To use gettimeofday, you modify your program to measure performance.
clock_gettime(CLOCK_MONOTONIC, ...)  

- gettimeofday() considered harmful: the measured time does not always run at the same speed. Sometimes time even goes backwards.  
- Use clock_gettime(CLOCK_MONOTONIC, ...).

```c
#include <time.h>

struct timespec start, end;
clock_gettime(CLOCK_MONOTONIC, &start);
function_to_measure();
clock_gettime(CLOCK_MONOTONIC, &end);
// Note: nanoseconds, not microseconds.
double tdiff = (end.tv_sec-start.tv_sec)
    + 1e-9*(end.tv_nsec-start.tv_nsec);
```
Instrumenting with a compiler

○ The compiler can instrument your code, e.g., for gprof:
  – Add -pg flag to your compiles and links.
  – Run your program.
  – Run gprof to get an analysis of where time was spent.

○ Gprof instrumentation causes your program to be interrupted to record the program counter 100 times per second.

○ Not accurate if you don’t get enough samples.
Fine-grained profiling

- The Performance API (PAPI)
  - [http://icl.cs.utk.edu/papi/overview/](http://icl.cs.utk.edu/papi/overview/)
  - A set of APIs to read the performance counters