Writing Efficient Programs

M. Anton Ertl
anton@mips.complang.tuwien.ac.at
http://www.complang.tuwien.ac.at/anton/

Institut für Computersprachen
Technische Universität Wien
Translated by David Gregg, Trinity College Dublin
Is efficiency important?

- Much software is fast enough
- Other software not yet fast enough
- More frequent invocation, Other processes
- Larger inputs
- Better functionality
Types of efficiency

• Running time
  – CPU
  – Hard disk access time
  – Network
  – Other I/O

• Memory
  – RAM
  – ROM
  – Disk
  – External storage
Costs of inefficiency

- Lost user time
- Other processes
- Unusability for real-time processing
- More expensive hardware
How much efficiency do we need?

- Response to user commands: 300ms
- Music: 20ms
- Animated software: screen refresh rate (12ms).
- Improve efficiency until other components dominate
- Often commercial trade-offs have to be made
Other Goals

• Correctness

• Clarity, simplicity

• Development effort/cost

• Maintainence effort/cost

• Time-to-market
Observations

• 80-20 Rule

• Predicting where the “hot spots” will be is unreliable

General approach

• First concentrate on good software engineer (simple, flexible, maintainable)

• Measure the resulting program

• Optimize the time-critical parts

Problem: Efficiency problems in the specification and design
Method

Start with unoptimized program

- Measure performance
- If too inefficient, profile
- Apply program transformation
- Test optimized program
- Repeat
Why doesn’t the compiler do this stuff?

The compiler does optimize the program, but

• must stay within the semantics of the programming language

• must avoid potential “pessimizations”

• only attempts transformations that can be identified quickly and with limited memory

• can only include optimizations that are used reasonably often

• there may be dependences between optimizations
Typical stumbling blocks for compilers

• Aliasing

• Side-effects, Exceptions

• Loops that execute zero times
Hardware Features

1cycle 3-4 independent instructions
1cycle Latency of ALU instruction
2-3cyc. Latency of Load (L1-Hit)
10cyc. Latency of Load (L1-Miss, L2-Hit)
100cyc. Latency of Load (L2-Miss, Main Memory)
30ns Transfer time for a cacheline (32-64B)
1cyc. correctly predicted branch
14cyc. branch misprediction
4-10cyc. latency of integer multiplication
 4cyc. Latency FP-add or mul
50cyc. Latency of division
100us IP-Ping over fast ethernet (100Mb/s)
100us 1KB transfer over fast ethernet
10ms Latency of disk access (seek + rotational delay)
10ms 400KB sequential access (ohne delay)
100ms Latency of CD-ROM (spinning; otherwise seconds)
100ms 600KB sequential CD-ROM access
Data structures und Algorithms

- Efficient implementation of an inefficient algorithm is (usually) a waste of time

- Ideally efficient implementation of an efficient algorithm

- Goals: simplicity, efficiency, flexibility

- Problem: Abstraction, abstract data types
Algorithmic Complexity

• Often considers the worst case

• Counts operations, not always relevant for running time

• Ignores constant factors

• Logarithmic factors
Programming language: Issues

- Pointer aliasing: C vs. Fortran
- Nested objects: Java vs. C++
- Scaling in pointer arithmetic: C vs. Forth
- Null terminated strings in C
- “C++ is slow” (Or Java is slow or Python).
- But slow for what?
Code motion out of loops

for (...) {
    ....
    a[i] = sin(x) * PI;  // some computation
    ...
}

The computation must have no side effects
The computation must depend on no results computed in the loop.

temp = sin(x) * PI;
for (...) {
    ...
    a[i] = temp;
    ...
}
Another example

```c
for (i = 0; i < strlen(s); i++) {
    ....

    ...
}
```

Can be rewritten

```c
length = strlen(s);
for (i = 0; i < length; i++) {
    ...
    ...
}
```
Combining Tests

For example, sentinel in search loop

```c
for (i=0; a[i]!=key && i<n; i++) {
    ...
}
```

can be written

```c
a[n] = key;
for (i=0; a[i]! = key; i++) {
    ...
}
```

Damages maintainability, reentrancy
Loop Unrolling

for (i=0; i<n; i++)
    body(i);

for (i=0; i<n-1; i+=2) {
    body(i);
    body(i+1);
}
for (; i<n; i++)
    body(i);

People can sometimes do a better job of optimizing unrolled code than the compiler
Unrolling to remove copies

old_a = a;
a = ...;
... = ... old_a ...;

Unrolling by a factor of 2

a2 = ...;
... = ... a1 ...;
a1 = ...;
... = ... a2 ...;
Software Pipelining

for (...) {
    a = ...;
    ... = ... a ...;
}

The computation of a must have no side-effects

a = ...;
for (...) {
    ... = ... a ...;
    a = ...;
}

Or if you need to keep the value of a you can write

new_a = ...;
for (...) {
    a = new_a;
    new_a = ...;
    ... = ... a ...;
}
Unconditional Branch Removal

while (test)
    code;

if (test)
    do
        code;
    while (test);

This transformation is trivial for the compiler, so there is no need to do it manually.
Loop Peeling

while (test)
    code;

if (test) {
    code;
    while (test)
        code;
}
**Loop Fusion**

```plaintext
for (i=0; i<n; i++)
    code1;
for (i=0; i<n; i++)
    code2;

Iteration $k$ in code2 does not depend on iteration $j < k$ in code1.

for (i=0; i<n; i++) {
    code1;
    code2;
}
```
Exploit Algebraic Identities

\sim a \& \sim b

\sim (a \mid b)

Computer arithmetic is neither integer arithmetic nor real-arithmetic (overflows, rounding errors with FP).
Short-circuiting Monotone Functions

for (i=0, sum=0; i<n; i++)
    sum += x[i];
flag = sum > cutoff;

Assuming all $x[i] \geq 0$, sum and i are not used again:

for (i=0, sum=0; i<n && sum <= cutoff; i++)
    sum += x[i];
flag = sum > cutoff;

Unrolling for fewer comparisons and branches
Long-circuiting

A && B

A and B compute flags, B has no side-effects

A & B

Use when B is cheap and A difficult to predict.
Arithmetic with Flags

```c
if (flag)
    x++;

x += (flag != 0);
```
Other Flag Representations

\[(a<0) \neq (b<0)\]

\[(a^b) < 0\]
Reordering Tests

A && B

A and B have no side effects

B && A

Which order of evaluation? First:

- Cheapest
- Most predictable
- höhere Abkürzwahrscheinlichkeit
Reordering Tests

if (A)
  ...
else if (B)
  ...

A and B have no side-effects, $\neg (A \land B)$

if (B)
  ...
else if (A)
  ...
Precompute Functions

int foo(char c)
{
    ...
}

foo() has no side-effects.

int foo_table[] = {...};

int foo(char c)
{
    return foo_table[c];
}
Boolean/State Variable Elimination

flag = exp();
S1;
if (flag)
  S2;
else
  S3;

flag is not used again after this.

if (exp()) {
  S1;
  S2;
} else {
  S1;
  S3;
}
Collapsing Procedure Hierarchies

- Inlining

- Specialization

```c
foo(int i, int j)
{
  ...
}

... foo(1, a);

foo_1(int j)
{
  ...
}
```
Exploit Common Cases

Handle all cases correctly and common cases efficiently.

- Memoization: For expensive functions: store already computed results.

- Pre-computed tables/code sequences for frequent parameters
Coroutines

Instead of multi-pass processing:

coroutine producer {
    for (...)            
        ... consumer(x); ...
}

coroutine consumer {
    for (...)            
        ... x = producer(); ...
}

Also pipelines, iterators, etc.
Transformations on Recursive Procedures

• Tail call optimization

• Inlining

• Ein rekursiver Aufruf: durch Zähler ersetzen

• Generally: use an explicit stack

• Für kleine Problemgrößen andere Methode

• Recursion instead of iteration for automatic cache-blocking
Tail Call Optimization

void traverse_simple( PNODE p )
{
    if ( p!=0 )
    {
        traverse_simple( p->l );
        ...
        traverse_simple( p->r );
    }
}

start:
    if ( p!=0 )
    {
        traverse_simple( p->l );
        ...
        p = p->r; goto start;
    }
Zählerverwendung

```c
foo()
{
    if (...) {
        code1;
        foo();
        code2;
    }
}
```

```c
while (...) {
    count++;
    code1;
}
```

```c
for (i=0; i<count; i++)
    code2;
```
Parallelism

- Between several CPUs: multithreading
- Between CPU and disk: prefetching, write buffering
- Between CPU and graphics card: triple buffering
- Between CPU and memory: prefetching
- Between machine instructions: instruction scheduling
- SIMD
Compile-Time Initialization

- Initialize tables at compile-time instead of at run-time

- CPU time vs. load time from disk
Strength Reduction/Incremental Algorithms

\[
y = x^2; \\
x += 1; \\
y = x^2; \\
y = x^2; \\
x += 1; \\
y += 2x-1;
\]
Common subexpression elimination

\[ a = \text{Exp}; \]
\[ b = \text{Exp}; \]

Exp has no side-effects

\[ a = \text{Exp}; \]
\[ b = a; \]
Pairing Computation

• Additional results little work

• E.g. Division and remainder; sin and cos
Exploit Word Parallelism/SIMD

for (count=0; x > 0; x >>= 1)
    count += x&1;

/* 64-bit-specific */
x = (x & 0x5555555555555555L) + ((x>>1) & 0x5555555555555555L);
x = (x & 0x3333333333333333L) + ((x>>2) & 0x3333333333333333L);
x = (x+(x>>4)) &0x0f0f0f0f0f0f0f0fL;
x = (x+(x>>8)) &0x001f001f001f001fL*;
x = (x+(x>>16)) &0x0000003f0000003fL*;
x = (x+(x>>32)) &0x7fL;
count = x;

0|0|0|1|1|0|1|1
  0| 1| 1| 2
    1| 3
      4
Data Structure Augmentation

• Fields with redundant data to accelerate particular operations

• Great danger of inconsistent data structures

• Hints, that may be correct, but do not have to be

• Memoization

• Caching
Lazy Evaluation

- Example: Finite-state automaton for regular expressions
Packing

- No unnecessary bytes/bits (bitfields in C, packed in Pascal)
- Data compression
- Code size
- Cache behaviour
Interpreters, Factoring

- Abstract similar code sections as procedures (functions)

- Schematische Programme per Interpreter implementieren
Compiler Flags

• Compiler optimization flags can give significant speedups

• Some flags give good speedups but have other downsides, so are not enabled by default (e.g. -fomit-frame-pointer in gcc on x86)

• A compiler like gcc provides dozens of optimization flags

• The right combination of flags often gives additional speedups

• The Acovea tool can be used to tune the flags for a particular program
Some Interesting GCC Flags

- `-fomit-frame-pointer`: Don't keep the pointer to the stack frame in a register if it is not needed.

- `-O0`: Optimize for code size rather than speed.

- `-fprofile-generate,-fprofile-use`: Generate profiling information on test run that helps the compiler make better decisions.

- `-fstrict-aliasing`: Compiler may assume the program follows strict pointer aliasing rules.

- `-fwhole-program`: tells the compiler that the current file is the whole program, so there will not be calls from other files.
Writing faster C code

• If function is only used in one file, declare it “static”

• Use “inline” declaration where you want function inlined

• Use “restrict” pointers to tell compiler about pointer aliasing

• Using “register” declaration on a variable will ensure that you don’t accidentally make it unavailable for register allocation

• If you use a value repeatedly in a loop, copy it into a local variable, where it becomes eligible for register allocation
Programming for locality

- Design data structures that maintain locality

- Arrays are good, linked lists are bad

- Hash tables are usually faster than binary trees

- In structs, declare the largest component items first

- Consider arrays of structures versus structures of arrays