Compiler Optimization

Mirko Köster

Seminar
Effiziente Programmierung in C
Fachbereich Informatik
Universität Hamburg

2012-11-29

Overview

Key Aspects

- What is the compiler capable of?
- What are its weaknesses?
- How can you make use of it?

Content

- Automatic Optimization
- Profile Guided Optimization
- Aiding Optimizations
- 'Safe' / 'Unsafe' Optimizations
- OpenMP

Overview

Key Aspects

- What is the compiler capable of?
- What are its weaknesses?
- How can you make use of it?

Content

- Automatic Optimization
- Profile Guided Optimization
- Aiding Optimizations
- 'Safe' / 'Unsafe' Optimizations
- OpenMP

Preface

Compiler

- Some examples are from the GNU C compiler
- There are lots of other good compilers available
- But I'll just give you an overview of the concepts
- Refer to the manual of your compiler for specific optimizations

Architecture

- In this presentation I'll focus on the x86 architecture
- If you are developing for another architecture get familiar with it (but the basic concepts will work there as well)

Preface

Compiler

- Some examples are from the GNU C compiler
- There are lots of other good compilers available
- But I'll just give you an overview of the concepts
- Refer to the manual of your compiler for specific optimizations

Architecture

- In this presentation I'll focus on the x86 architecture
- If you are developing for another architecture get familiar with it (but the basic concepts will work there as well)

Automatic Optimization

Definition

- Changes that don't affect the result
- May optimize
  - Execution speed
  - File size of the executable
  - or even power consumption
- activated by compiler options / flags

How does it work?

1. Analyse source code
2. Assume stricter rules as the c-language
3. Prove assumptions
4. Apply optimization(s)

Automatic Optimization

Definition

- Changes that don't affect the result
- May optimize
  - Execution speed
  - File size of the executable
  - or even power consumption
- activated by compiler options / flags

How does it work?

1. Analyse source code
2. Assume stricter rules as the c-language
3. Prove assumptions
4. Apply optimization(s)

Automatic Optimization

How to use it

- activated by -0[level]
- or manually by the specific flag

-o1

- -fauto-inc-dec
- -fcompare-elim
- -fcprop-registers
- -fdce
- -fdefer-pop
- -fdelayed-branch
- -fdse
- -fguess-branch-probability
- -fif-conversion2
- -fif-conversion

-o1

- -fipa-pure-const
- -fipa-profile
- -fipa-reference
- -fmerge-constants
- -fsplit-wide-types
- -ftree-bit-ccp
- -ftree-builtin-call-dce
- -ftree-ccp
- -ftree-ch
- -ftree-copyrename
- -ftree-dce

-o1

- -ftree-dominator-opts
- -ftree-dse
- -ftree-forwprop
- -ftree-fre
- -ftree-phiprop
- -ftree-slsr
- -ftree-sra
- -ftree-pta
- -ftree-ter
- -funit-at-a-time

Automatic Optimization

How to use it

- activated by -0[level]
- or manually by the specific flag

-o1

- -fauto-inc-dec
- -fcompare-elim
- -fcprop-registers
- -fdce
- -fdefer-pop
- -fdelayed-branch
- -fdse
- -fguess-branch-probability
- -fif-conversion2
- -fif-conversion

-o1

- -fipa-pure-const
- -fipa-profile
- -fipa-reference
- -fmerge-constants
- -fsplit-wide-types
- -ftree-bit-ccp
- -ftree-builtin-call-dce
- -ftree-ccp
- -ftree-ch
- -ftree-copyrename
- -ftree-dce

-o1

- -ftree-dominator-opts
- -ftree-dse
- -ftree-forwprop
- -ftree-fre
- -ftree-phiprop
- -ftree-slsr
- -ftree-sra
- -ftree-pta
- -ftree-ter
- -funit-at-a-time

Automatic Optimization

-o2 (includes all from -01)

- -fthread-jumps
- -falign-functions -falign-jumps
- -falign-loops -falign-labels
- -fcaller-saves
- -fcrossjumping
- -fcse-follow-jumps -fcse-skip-blocks
- -fdelete-null-pointer-checks
- -fdevirtualize
- -fexpensive-optimizations
- -fgcse -fgcse-lm
- -fhoist-adjacent-loads
- -finline-small-functions
- -findirect-inlining

-o2 (includes all from -01)

- -fipa-sra
- -foptimize-sibling-calls
- -fpartial-inlining
- -fpeephole2
- -fregmove
- -freorder-blocks -freorder-functions
- -frerun-cse-after-loop
- -fsched-interblock -fsched-spec
- -fschedule-insns -fschedule-insns2
- -fstrict-aliasing -fstrict-overflow
- -ftree-switch-conversion -ftree-tail-merge
- -ftree-pre
- -ftree-vrp

Automatic Optimization

-o3 (includes all from -02)

- -finline-functions
- -funswitch-loops
- -fpredictive-commoning
- -fgcse-after-reload
- -ftree-vectorize
- -fvect-cost-model
- -ftree-partial-pre
- -fipa-cp-clone

-o0 (default)

Reduce compilation time and make debugging produce the expected results

-os (Optimize for size)

disables

- -falign-functions
- -falign-jumps
- -falign-loops
- -falign-labels
- -freorder-blocks
- -freorder-blocks-and-partition
- -fprefetch-loop-arrays
- -ftree-vect-loop-version

Automatic Optimization

Some optimizations are very time-consuming

- Some problems are np hard
- Some problems are even undecidable
- Tradeoff: in those cases the compiler won't give the optimal result but a good result (to save time/space during compilation)

Architecture Independent Optimizations

Definition

- Do not rely upon knowledge of the underlying architecture
- Can be applied under any circumstances after the assumptions have been proven

Loop Invariant Code Motion

Definition

Moves code out of a loop if it is invariant of the loop variable

unoptimized

```
int sum=0, x;
for(int i = 0; i < n; i++) {
  sum += i;
  x = 5;
}
```

optimized

```
int sum=0, x = 5;
for(int i = 0; i < n; i++) {
  sum += i;
}
```

Loop Invariant Code Motion

Definition

Moves code out of a loop if it is invariant of the loop variable

unoptimized

```
int sum=0, x;
for(int i = 0; i < n; i++) {
  sum += i;
  x = 5;
}
```

optimized

```
int sum=0, x = 5;
for(int i = 0; i < n; i++) {
  sum += i;
}
```

Const Propagation (with Loop Optimization)

Definition

Evaluation of expressions with known values at compile time

unoptimized

```c
int N = 10, sum = 0;
for(int i = 0; i < N; i++)
  sum += i;

printf("sum = %d\n", sum);
```

optimized

```c
int sum = 0;
for(int i = 0; i < 10; i++)
  sum += i;

printf("sum = %d\n", sum);
```

Const Propagation (with Loop Optimization)

Definition

Evaluation of expressions with known values at compile time

unoptimized

```c
int N = 10, sum = 0;
for(int i = 0; i < N; i++)
  sum += i;

printf("sum = %d\n", sum);
```

optimized

```c
int sum = 0;
for(int i = 0; i < 10; i++)
  sum += i;

printf("sum = %d\n", sum);
```

Const Propagation (with Loop Optimization)

Definition

Evaluation of expressions with known values at compile time

unoptimized

```c
int N = 10, sum = 0;
for(int i = 0; i < N; i++)
  sum += i;

printf("sum = %d\n", sum);
```

optimized

```c
printf("sum = %d\n", 45);
```

Dead Code Elimination

Definition

Removes code that is unnecessary or never executed

unoptimized

```
unsigned int x = foobar();
if(x < 0) {
  printf("never executed\n");
} else {
  printf("x: %u\n", x);
}
```

optimized

```
unsigned int x = foobar();
if(x >= 0) {
  printf("x: %u\n", x);
}
```

Dead Code Elimination

Definition

Removes code that is unnecessary or never executed

unoptimized

```c
unsigned int x = foobar();
if(x < 0) {
  printf("never executed\n");
} else {
  printf("x: %u\n", x);
}
```

optimized

```c
unsigned int x = foobar();
if(x >= 0) {
  printf("x: %u\n", x);
}
```

Dead Code Elimination

Definition

Removes code that is unnecessary or never executed

unoptimized

```c
unsigned int x = foobar();
if(x < 0) {
  printf("never executed\n");
} else {
  printf("x: %u\n", x);
}
```

optimized

```c
printf("x: %u\n", foobar());
```

Common Subexpression Elimination

Definition

Reduces occurences of multiple common subexpressions

unoptimized

```
void foo(int *a, int n) {
  for(int i = 0; i < n; i++)
    a[i] += a[i]/n + a[i]*n;
}
```

optimized

```
void foo(int *a, int n) {
  int temp;
  for(int i = 0; i < n; i++)
    temp = a[i]
    a[i] += temp/n + temp*n;
}
```

Common Subexpression Elimination

Definition

Reduces occurences of multiple common subexpressions

unoptimized

```c
void foo(int *a, int n) {
  for(int i = 0; i < n; i++)
    a[i] += a[i]/n + a[i]*n;
}
```

optimized

```c
void foo(int *a, int n) {
  int temp;
  for(int i = 0; i < n; i++)
    temp = a[i]
    a[i] += temp/n + temp*n;
}
```

Interprocedural Optimization

Definition

looks at multiple functions and how they work together

Arguments in Registers

- passing arguments in registers instead of pushing/popping them to/from stack
- reduces call/return overhead
- requires modification of caller and callee

Inlining

Definition

- For small functions the overhead of calling may be larger in relation to the body.
- Inlining replaces the call to the function with the body.

unoptimized

```c
int foo(int a) {
  return a * (a+1);
}
...
int a[5];
for(int i = 0; i < 5; i++)
  a[i] = foo(i);
```

optimized

```c
int a[5];

a[0] = 0 * 1;
a[1] = 1 * 2;
a[2] = 2 * 3;
a[3] = 3 * 4;
a[4] = 4 * 5;
```

Inlining

Definition

- For small functions the overhead of calling may be larger in relation to the body.
- Inlining replaces the call to the function with the body.

unoptimized

```
int foo(int a) {
  return a * (a+1);
}
...
int a[5];
for(int i = 0; i < 5; i++)
  a[i] = foo(i);
```

optimized

```
int a[5];

a[0] = 0 * 1;
a[1] = 1 * 2;
a[2] = 2 * 3;
a[3] = 3 * 4;
a[4] = 4 * 5;
```

Interprocedural Constant Propagation

Definition

Evaluation of expressions with known values at compile time taking multiple functions into account

unoptimized

```c
static int square(int x) {
  return x*x;
}

printf("5^2=%d\n",square(5));
```

optimized

```c
static int square(int x) {
  return x*x;
}

printf("5^2 = %d\n", 25);
```

Interprocedural Constant Propagation

Definition

Evaluation of expressions with known values at compile time taking multiple functions into account

unoptimized

```c
static int square(int x) {
  return x*x;
}

printf("5^2=%d\n",square(5));
```

optimized

```c
static int square(int x) {
  return x*x;
}

printf("5^2 = %d\n", 25);
```

Architecture Dependent Optimizations

Definition

- Target-specific optimizations
- The compiler has to know the target architecture
- caution: the executable may not run on older machines

What makes a target architecture?

- Instruction set (e.g. x86)
- Number of (special purpose) registers
- Cache size & type
- possibly some instruction set extensions (MMX, SSE...)

Architecture Dependent Optimizations

Definition

- Target-specific optimizations
- The compiler has to know the target architecture
- caution: the executable may not run on older machines

What makes a target architecture?

- Instruction set (e.g. x86)
- Number of (special purpose) registers
- Cache size & type
- possibly some instruction set extensions (MMX, SSE...)

Instruction Set

Overview History

- 1985 x86 32bit
- 1989 x87 FPU (Co-Processor)
- 1993 MMX
- 1997 SSE, 3DNow!
- 2000 SSE2
- 2003 x86-64 64bit
- 2004 SSE3
- 2007 SSE4a
- 2011 SSE5/AVX
- 2013 AVX2, FMA3

-mtune & -march

-mtune

This option optimizes for the given architecture, making the code faster on those machines. But it will still run on other architectures.

-march

This option will make the most of the given architecture. May not run on other architectures.

example options

- i386
- pentium
- corei7
- amdfam10

gcc: 32 vs 64 Bit

Advantage of compiling for 64bit machines

- The compiler can make use of
  - at least MMX, SSE and SSE2, since every x86-64 machine supports these.
  - 16 registers (64 bit) instead of 8 registers (32 bit)
  - larger virtual address space (at least 48 bit = 256 TiB)

Automatic Vectorization

Definition

The compiler makes use of SIMD

source

```
float a[128];
...
for(int i=0; i < 128; i++)
  a[i] *= 2.5f;
```

Example Optimization using AVX

- Width of SIMD registers: 256bit
- Float uses 32bit
- -> 8 calculations in parallel
- 16 * 8 simultaneous multiplications instead of 128 in sequence

Automatic Vectorization

Definition

The compiler makes use of SIMD

source

```
float a[128];
...
for(int i=0; i < 128; i++)
  a[i] *= 2.5f;
```

Example Optimization using AVX

- Width of SIMD registers: 256bit
- Float uses 32bit
- -> 8 calculations in parallel
- 16 * 8 simultaneous multiplications instead of 128 in sequence

Profile Guided Optimization

Definition

The execution of the program is profiled, so the compiler can learn from the 'behaviour' of the code

Steps

- compile and link it with profiling enabled
- run the program - make sure all the time-critical parts are executed
- profiling data will be written to disk
- recompile making use of the profiling data

Function Ordering

Definition

Re-orders functions to improve instruction cache hit rate

unoptimized

```
int foo() {
  ... //several lines of code
}
float someFunction() {
  ... //several lines of code
}
... //more functions
int bar() {
  ... //several lines of code
}
```

optimized

```
int foo() {
  ... //several lines of code
}
int bar() {
  ... //several lines of code
}
float someFunction() {
  ... //several lines of code
}
... //more functions
```

Function Ordering

Definition

Re-orders functions to improve instruction cache hit rate

unoptimized

```
int foo() {
  ... //several lines of code
}
float someFunction() {
  ... //several lines of code
}
... //more functions
int bar() {
  ... //several lines of code
}
```

optimized

```
int foo() {
  ... //several lines of code
}
int bar() {
  ... //several lines of code
}
float someFunction() {
  ... //several lines of code
}
... //more functions
```

Basic Block Ordering

Definition

- Similar to function ordering
- Same goal: improve instruction cache hit rate
- Re-orders blocks

C-Compiler

Mirko Köster

Automatic Optimization
Architecture Independent
Inter-Procedural
Architecture Dependent
Profile Guided Optimization
Aiding Optimizations
'Safe' / 'Unsafe' Optimizations
OpenMP
Conclusion
Questions?
Sources

Switch Statement Optimization

Definition

Sorts the cases in a switch statement by frequency of execution

unoptimized

```
switch(expression)
{
    case constant1:
        statements; break;
    case constant2:
        statements; break;
    case constant3:
        statements; break;
    default:
        statements;
}
```

optimized

```
switch(expression)
{
    case constant3:
        statements; break;
    case constant1:
        statements; break;
    case constant2:
        statements; break;
    default:
        statements;
}
```

Switch Statement Optimization

Definition

Sorts the cases in a switch statement by frequency of execution

unoptimized

```
switch(expression)
{
    case constant1:
        statements; break;
    case constant2:
        statements; break;
    case constant3:
        statements; break;
    default:
        statements;
}
```

optimized

```
switch(expression)
{
    case constant3:
        statements; break;
    case constant1:
        statements; break;
    case constant2:
        statements; break;
    default:
        statements;
}
```

Improved Register Allocation

Definition

Keeps the locally most frequently used variables in registers

note

The problem of register allocation is np-hard without profiling

Aiding Optimizations

Why this is useful

- The compiler 'enforces rules of the C-Standard' to ensure correct programs
- Often the compiler has to make conservative assumptions
- If it had more knowledge about the code, it could optimize more aggressively
- The programmer can help the compiler

Data Layout

Definition

A good data layout uses memory space and cache more efficiently

unoptimized

```c
struct foo {
  char a;
  float x[8];
  char b;
  float y[8];
  char c;
  float z[8];
};
```

optimized

```c
struct foo {
  float x[8];
  float y[8];
  float z[8];
  char a;
  char b;
  char c;
};
```

Data Layout

Definition

A good data layout uses memory space and cache more efficiently

unoptimized

```
struct foo {
  char a;
  float x[8];
  char b;
  float y[8];
  char c;
  float z[8];
};
```

optimized

```
struct foo {
  float x[8];
  float y[8];
  float z[8];
  char a;
  char b;
  char c;
  };
```

Pragma Vector Aligned

Definition

- Communicates data layout information to the compiler
- Some architectures contain instructions that execute faster if the data is guaranteed to be aligned on specific memory boundaries

source

```
float a[128];
...
#pragma vector aligned
for(int i=0; i < 128; i++)
  a[i] *= 2.5f;
```

options

- aligned
- unaligned
- always

Pragma Vector Aligned

Definition

- Communicates data layout information to the compiler
- Some architectures contain instructions that execute faster if the data is guaranteed to be aligned on specific memory boundaries

source

```c
float a[128];
...
#pragma vector aligned
for(int i=0; i < 128; i++)
  a[i] *= 2.5f;
```

options

- aligned
- unaligned
- always

'Safe' / 'Unsafe' Optimizations

'normal' behaviour

- Most optimizations won't change the result of computations
- especially not the -o[level] options
- the compiler is conservative

more optimizations

- compiler options that might change the results
- but the computations may be faster
- caution: only use them if you don't need the precision
- e.g. -ffast-math

'Safe' / 'Unsafe' Optimizations

'normal' behaviour

- Most optimizations won't change the result of computations
- especially not the -o[level] options
- the compiler is conservative

more optimizations

- compiler options that might change the results
- but the computations may be faster
- caution: only use them if you don't need the precision
- e.g. -ffast-math

OpenMP

Definition

Shared-Memory Multithreading Programming Interface

unoptimized

```c
void foobar(int *a, int n) {
  for (int i = 0; i < n; i++)
    a[i] = 2 * i;
}
```

optimized

```c
void foobar(int *a, int n) {
  #pragma omp parallel for
  for (int i = 0; i < n; i++)
    a[i] = 2 * i;
}
```

OpenMP

Definition

Shared-Memory Multithreading Programming Interface

unoptimized

```
void foobar(int *a, int n) {
  for (int i = 0; i < n; i++)
    a[i] = 2 * i;
}
```

optimized

```
void foobar(int *a, int n) {
  #pragma omp parallel for
  for (int i = 0; i < n; i++)
    a[i] = 2 * i;
}
```

What we've learned today

Remember

- Which optimizations the compiler can do by himself -> **readability over manual optimization**
- Tell the compiler details about the destination architecture
- Where the compiler needs some help (aided optimization)
- Optimize manually, where the compiler can't help - but only if you can expect a real performance impact.
  if not -> **readability over manual optimization**
- or: "Premature Optimization is the root of all evil"

Questions?

Thank you for your attention

Questions?

Sources

Resources I used to prepare this presentation

- http://en.wikipedia.org/wiki/Optimizing_compiler
- http://gcc.gnu.org/onlinedocs/gcc-4.7.2/gcc/Optimize-Options.html
- http://gcc.gnu.org/onlinedocs/gcc/i386-and-x86_002d64-Options.html
- http://www.embedded.com/design/mcus-processors-and-socs/4008892/Tuning-C-C--compilers-for-optimal-parallel-