# Loop Optimizations in Modern C Compilers

Chae Jubb

24 November 2014

# Loops are Important!

- Most of program execution spent here!
- Small inefficiencies are magnified with hundreds or thousands of iterations
- Deeply nested loops can contain many jump instructions

# Motivating Examples

```
int *fill_array(uint16_t in) {
    int *a =
        malloc(sizeof *a * in * 4);
    int i, marker;
    for (i = 0; i < in; ++i) {
         marker = 4 * i;
         a[marker + 0] = in % 2;
         a[marker + 1] = in % 3;
         a[marker + 2] = in % 5;
         a[marker + 3] = in % 7;
    }
    return a;
}
```

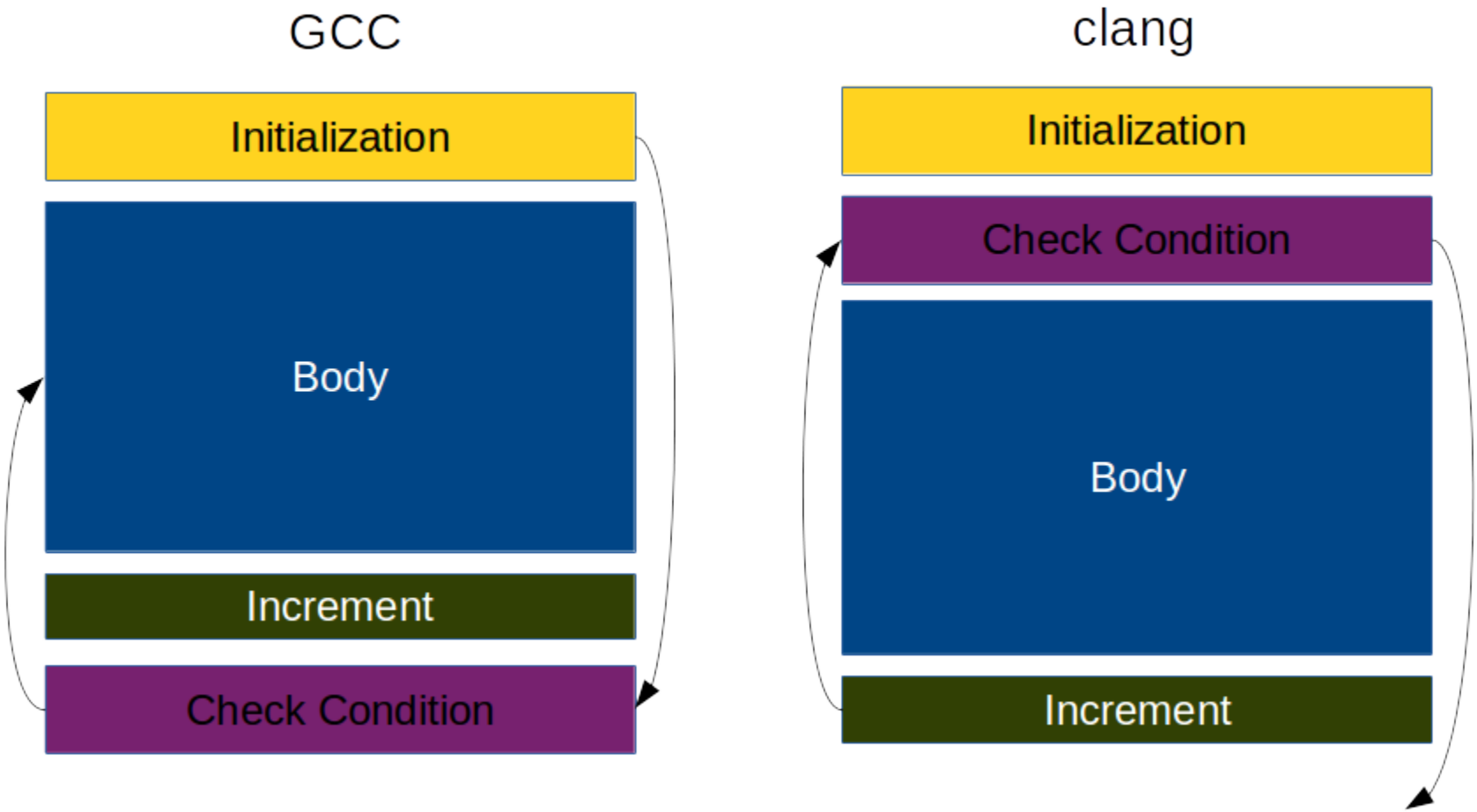
```
int count_zeros(uint32_t a) {
  unsigned int counter = 0;
  unsigned int i;
  for (i = 0; i < 32; ++i, a >>= 1)
  {
    counter += !(a & 0x00000001);
  }

  return counter;
}
```

# Fill Array

```
int *fill_array(uint16_t in) {
  int *a =
    malloc(sizeof *a * in * 4);
  int i, marker;
  for (i = 0; i < in; ++i) {
    marker = 4 * i;
    a[marker + 0] = in % 2;
    a[marker + 1] = in % 3;
    a[marker + 2] = in % 5;
    a[marker + 3] = in % 7;
  }
  return a;
}
```

- Clearly Inefficient
- *Induction Variable*
- *Loop Constants*
- Opportunity for Parallelism

# Unoptimized Loops: GCC vs. Clang



# Structure of Unoptimized Code

```
int *fill_array(uint16_t in) {
  int *a =
    malloc(sizeof *a * in * 4);
  int i, marker;
  for (i = 0; i < in; ++i) {
    marker = 4 * i;
    a[marker + 0] = in % 2;
    a[marker + 1] = in % 3;
    a[marker + 2] = in % 5;
    a[marker + 3] = in % 7;
  }
  return a;
}
```

- Translates statement by statement as expected
- gcc and clang implement modulus differently
  - clang: fewer instructions, takes twice as long as gcc

# Let's Optimize! (-O1)

```
int *fill_array(uint16_t in) {
  int *a =
    malloc(sizeof *a * in * 4);
  int i, marker;
  for (i = 0; i < in; ++i) {
    marker = 4 * i;
    a[marker + 0] = in % 2;
    a[marker + 1] = in % 3;
    a[marker + 2] = in % 5;
    a[marker + 3] = in % 7;
  }
  return a;
}
```

- "marker" variable disappears
  - replaced by x4 in addressing
- Loop Inversion!
  - If wrapping a do-while

```
test   ebx,ebx <ebx holds max ecx value>
je     40074e <after end of loop>
 <body of loop>
inc    ecx
cmp    ecx,ebx
jl     400740 <body>
```

# Why clang is awesome!

```
int *fill_array(uint16_t in) {
  int *a =
    malloc(sizeof *a * in * 4);
  int i, marker;
  for (i = 0; i < in; ++i) {
    marker = 4 * i;
    a[marker + 0] = in % 2;
    a[marker + 1] = in % 3;
    a[marker + 2] = in % 5;
    a[marker + 3] = in % 7;
  }
  return a;
}
```

- Uses XMM registers (at O1)
  - 128 bits wide (four 4 byte integers)
- Loads set of 4 into `xmm0` and repeatedly writes to array
- As good as it gets!
  - 25x faster than -O0!

# Further Optimization (-O2 and -O3)

```
int *fill_array(uint16_t in) {
  int *a =
    malloc(sizeof *a * in * 4);
  int i, marker;
  for (i = 0; i < in; ++i) {
    marker = 4 * i;
    a[marker + 0] = in % 2;
    a[marker + 1] = in % 3;
    a[marker + 2] = in % 5;
    a[marker + 3] = in % 7;
  }
  return a;
}
```

- clang
  - done after -O1
- gcc
  - extracts loop constants at -O2
  - finally uses XMM at -O3

# Counting Zeros

**Original**

```
for (i = 0; i < 32; ++i, a >>= 1) {
    counter += !(a & 0x00000001);
}
```

**Alternate 1: Branch**

```
for(i = 0; i < 32; ++i, a >>= 1) {
    if ((a & 0x00000001) == 0)
        counter++;
}
```

**Alternate 2: Nested For**

```
for (j = 0; j < 4; ++j) {
  for (i = 0; i < 8; ++i, a >>= 1) {
    counter += !(a & 0x00000001);
  }
}
```

- Three versions
  - Branchless
  - Branch
  - Nested For
- Each handled differently
- Both compilers emit expected code at -O0

# Boring Case (Original)

- Both compilers do the same thing at -O1 and make no more changes at -O2 or -O3
- Eliminate stack
- `dec` rather than `inc` for loop counter
  - Saves an instruction

```
mov     ecx,0x20
 <body of loop>
sub     ecx,0x1

jne     4005e7 <body>
```

```
mov     ecx,0x0
 <body of loop>
add     ecx,0x1
cmp     ecx,0x1f
jbe     4005d4 <body>
```

# The Genius of Clang! (Branching)

- gcc keeps this case distinct
  - -O1: `cmp` followed by `adc`
    - (faster than -O2 and -O3 sometimes)
  - -O3: `test` followed by `cmove`
    - Still using processor flags
- clang converts to non-branching case!
  - Exact same emitted code as non-branching version for -O1, -O2, -O3

# GCC is good, too! (Nested For)

- gcc handles this case better for mild optimizations
  - Both loops with `dec` counters
- At high optimizations, clang performs
  - Complete loop unroll
    - All 32 iterations
  - gcc only unrolls inner loop (x8)
    - Very comparable to clang's full unroll

# Clang Drawbacks

- Completely unoptimized is outperformed by gcc
  - Does not fare well on “fill_array” example
  - Nearly 90% longer execution time on new computer
    - i7 quad-core
  - Nearly 70% longer execution time on older computer
    - Pentium 4 (how important really?)
- Sometimes uses `jmp` to `jmp` or `jmp` to next instruction
  - Just...why?

# Loop Optimizations in x86

- IMPORTANT!
- Loop Inversion
- Loop Invariants
- `inc` to `dec`
- Parallelization
- Branching to non-branching