



GCC Compiler Optimizations



Optimization: Making your program **faster or more efficient** with space or time!

GCC can perform a variety of optimizations for us, when compiling with different flags
 ► (–O0 for none, –O2 for nearly all reasonable optimizations, –O3 is more aggressive)

Constant Folding: Compiler can pre-compute constant values, including e.g., constant arithmetic, `sizeof(int) → 4`, `strlen("AliceInWonderland") → 17`

Common subexpression elimination: Avoids recalculating same result multiple times

Strength reduction: Avoids multiplying/dividing by shifting and adding instead

Dead code elimination: If a piece of code can never be reached (e.g., code in a function after a return statement), the compiler can just remove it

Code motion: Rearrange code for better performance

Loop unrolling: Avoid many expensive conditions and jumps by copy-pasting the loop body (i.e. less iterations, with more work per iteration)

Tail Recursion: Alters some recursive patterns to be iterative instead

Callgrind!
 Because we all
 loved Valgrind
 so much



Callgrind is a profiling tool: observes an executing program to gather detailed statistics about resource consumption, e.g., time spent, instructions executed, memory usage, to **evaluate program efficiency**



Callgrind is part of the valgrind suite of tools, and **useful for our heap allocators** (we want to eventually make them as efficient as possible)



```
bbyan@myth61:~/CS107_TA/assignments/lab7$ valgrind --tool=callgrind --toggle-collect='*power*' ./trials
==3252945== Callgrind, a call-graph generating cache profiler
==3252945== Copyright (C) 2002-2017, and GNU GPL'd, by Josef Weidendorfer et al.
==3252945== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==3252945== Command: ./trials
==3252945==
==3252945== For interactive control, run 'callgrind_control -h'.
==3252945==
==3252945== Events      : Ir
==3252945== Collected : 427
==3252945==
==3252945== I refs:      427
bbyan@myth61:~/CS107_TA/assignments/lab7$ callgrind_annotate --auto=yes callgrind.out.3252945
```

Results saved to this file!



Callgrind vs Callgrind Annotate



Q: callgrind and callgrind_annotate, what's the difference?

Callgrind is a profiler: it gathers raw statistics on an executing program—and writes them to a file

Think: A measuring device / instrument

callgrind_annotate is an annotator / analyzer: it separately reads the file or report created by callgrind, and **prints useful info, e.g., annotates code with # of dynamic instructions per line. Lines erupting with many instructions are hotspots**

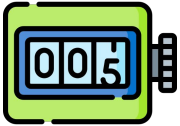
Think: Bar charts / data visualization

```
bbyan@myth61:~/CS107_TA/assignments/lab7$ valgrind --tool=callgrind --toggle-collect='*power*' ./trials
==3252945== Callgrind, a call-graph generating cache profiler
==3252945== Copyright (C) 2002-2017, and GNU GPL'd, by Josef Weidendorfer et al.
==3252945== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==3252945== Command: ./trials
==3252945==
==3252945== For interactive control, run 'callgrind_control -h'.
==3252945==
==3252945== Events      : Ir
==3252945== Collected : 427
==3252945==
==3252945== I refs:      427
bbyan@myth61:~/CS107_TA/assignments/lab7$ callgrind_annotate --auto=yes callgrind.out.3252945
```

```
. // Use the built in library function
1 long two_to_power_A(unsigned int exp) {
7   return pow(2, exp);
126 => /build/glibc-LcI20x/glibc-2.31/math/../w_pow_tem
2 }
.
. // is specialized 2^exp less work than general pow
1 long two_to_power_B(unsigned int exp) {
6   return exp2(exp);
42 => /build/glibc-LcI20x/glibc-2.31/math/../sysdeps/
2 }
```



Use Callgrind to Measure # of Instructions Executed



Callgrind writes detailed info to a file named **callgrind.out. [pid]**
 ➤ Here, [pid] is 3252945

Program being executed (**./trials**), and **--toggle-collect** to examine all functions with **power** in the name

```
bbyan@myth61:~/CS107_TA/assignments/lab7$ valgrind --tool=callgrind --toggle-collect='*power*' ./trials
==3252945== Callgrind, a call-graph generating cache profiler
==3252945== Copyright (C) 2002-2017, and GNU GPL'd, by Josef Weidendorfer et al.
==3252945== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==3252945== Command: ./trials
==3252945==
==3252945== For interactive control, run 'callgrind_control -h'.
==3252945==
==3252945== Events      : Ir
==3252945== Collected : 427
==3252945==
==3252945== I    refs:    427
```

Let's go take a look inside that **.out** file that callgrind just wrote to!

Total count of instructions executed when running program!

Read Callgrind Out File with Annotator!



We use **callgrind_annotate** to print out key information from the output file **callgrind.out. [pid]**, where [pid] is 3252945 here.

```
bbyan@myth61:~/CS107_TA/assignments/lab7$ callgrind_annotate --auto=yes callgrind.out.3252945
Profile data file 'callgrind.out.3252945' (creator: callgrind-3.15.0)
```

Note: The **=>** mark separates the:

- Count of instructions for **setting up + calling a function** (here, 7)
- Count of instructions for **executing the function itself** (here, 126)

Sample Annotated Printout

```
. // Use the built in library function
1 long two_to_power_A(unsigned int exp) {
7     return pow(2, exp);
126 => /build/glibc-LcI20x/glibc-2.31/math/
2 }
.
```

Instruction Counts: Static vs Dynamic



Static instruction count is the number of written assembly instructions

Dynamic instruction count is number of executed instructions **when a program is actually run** (note that a written line can be run multiple times, e.g., loops).

Example: This code computes 2^n by repeatedly multiplying with 2 inside a loop

```
mov    $0x1,%eax
mov    $0x1,%edx
cmp    %edi,%eax
ja     40117c <two_to_power_C+0x16>
add    %rdx,%rdx
add    $0x1,%eax
jmp    401170 <two_to_power_C+0xa>
mov    %rdx,%rax
retq
```

```
. long two_to_power_C(unsigned int exp) {
1     long result = 1;
187    for (int i = 1; i <= exp; i++) {
46        result *= 2;
.    }
.    return result;
2 }
```

Static count (lines of assembly) is just 9 instructions, so doesn't look too bad.

But when actually run, dynamic count is >200! The lines in the **loop** run many many times!

Takeaway: Dynamic counts are a better gauge of performance. Occasionally, the assembly code emitted with optimization can feature more static lines of assembly than the original, but that doesn't mean more instructions are actually executed!

