About GCC printf optimization

Peter Seiderer

November 21, 2005

Abstract

As you may have noticed, your GCC compiled source code is not always calling the functions you wrote in the source code. This can easily observed with printf in the *hello world* example. Starting from this some more GCC optimizations will be examined and analyzed in detail.

1 Introduction

For a start we examine what the *GCC* compiler [GCCa] will do with the *hello world* example when different optimization level are used.

```
#include <stdio.h>
int main(int argc, char *argv[]) {
  printf("hello world\n");
  return 0;
}
```

Figure 1: example1.c

Take a look at the *nm*[1] output when compiled without optimization.

```
> gcc-3.3.6 -O0 example1.c
> nm --undefined-only a.out
         w __gmon_start__
         w _Jv_RegisterClasses
         U __libc_start_main@@GLIBC_2.0
         U printf@@GLIBC_2.0
```

As expected the dynamically linked executable *a.out* will call the function *printf*. Now we compile the same source code with optimization on.

```
> gcc-3.3.6 -O1 example1.c
> nm --undefined-only a.out
         w __gmon_start__
```

[1] nm - list symbols from object files

```
         w _Jv_RegisterClasses
         U __libc_start_main@@GLIBC_2.0
         U puts@@GLIBC_2.0
```

The *nm* output indicates that *puts* is called instead of the original printf from the source code. To verify this we take a look at the intermediate assembler listing. First with optimization off.

```
> gcc-3.3.6 -O0 -S example1.c
```

Here the resulting *example1.s* file:

```
        .file   "example1.c"
        .section .rodata
.LC0:
        .string "hello world\n"
        .text
.globl main
        .type   main, @function
main:
        pushl   %ebp
        movl    %esp, %ebp
        subl    $8, %esp
        andl    $-16, %esp
        movl    $0, %eax
        subl    %eax, %esp
        movl    $.LC0, (%esp)
        call    printf
        movl    $0, %eax
        leave
        ret
        .size   main, .-main
        .ident  "GCC: (GNU) 3.3.6"
```

Second with optimization on.

```
> gcc-3.3.6 -O1 -S example1.c
```

And the resulting *example1.s* file:

```
        .file   "example1.c"
        .section .rodata
.LC0:
        .string "hello world"
        .text
.globl main
        .type   main, @function
main:
        pushl   %ebp
```

```
10          movl    %esp, %ebp
11          subl    $8, %esp
12          andl    $-16, %esp
13          movl    $.LC0, (%esp)
14          call    puts
15          movl    $0, %eax
16          movl    %ebp, %esp
17          popl    %ebp
18          ret
19          .size   main, .-main
20          .ident  "GCC: (GNU) 3.3.6"
```

The string constant in line 4 changed from *"hello world\n"* to *"hello world"*. The *call printf* on line 16 changed to *call puts* on line 14.

2 Optimization according to the source code

Where is the optimization implemented in the GCC source code:

- GCC-3.3.6: gcc/c-common.c line 1324
- GCC-3.4.4: gcc/builtins.c line 4651
- GCC-4.0.2: gcc/builtins.c:4654.

Step by step guide along the source code lines (for GCC-3.4.4). In the following examples we will omit the first two lines

```
        w __gmon_start__
        w _Jv_RegisterClasses
```

of the *nm* output.

2.1 No optimization if return code is used

No optimization if the return code of printf is evaluated [line 4662-4664].

```
#include <stdio.h>
int main(int argc, char *argv[]) {
  return printf("hello world\n");
}
```

Figure 2: example2.c

```
> gcc-3.4.4 -O1 example2.c
> nm --undefined-only a.out
         U __libc_start_main@@GLIBC_2.0
         U printf@@GLIBC_2.0
```

2.2 Format string must be a literal constant

The format string must be a literal string constant [line 4666-4677].

```
#include <stdio.h>
static const char *f() {
  return "hello world\n";
}
int main(int argc, char *argv[]) {
  printf(f());
  return 0;
}
```

Figure 3: example3.c

```
> gcc-3.4.4 -O1 example3.c
> nm --undefined-only a.out
         U __libc_start_main@@GLIBC_2.0
         U printf@@GLIBC_2.0
```

Note that the GCC-4.0.2 is smart enough to optimize this example if an optimization level greater or even $-O2$ is given.

```
> gcc-4.0.2 -Wall -O2 example3.c
 nm --undefined-only a.out
         U __libc_start_main@@GLIBC_2.0
         U puts@@GLIBC_2.0
```

2.3 ”$\%s\backslash n$”

A $printf$ call with the format string ”$\%s\backslash n$” [line 4679-4687] is converted to a $puts()$ call.

```
printf("%s\n", "hello world"); // converted to puts("hello world");
```

```
#include <stdio.h>
int main(int argc, char *argv[]) {
  printf("%s\n", "hello world");
  return 0;
}
```

Figure 4: example4.c

```
> gcc-3.4.4 -O1 example4.c
> nm --undefined-only a.out
         U __libc_start_main@@GLIBC_2.0
         U puts@@GLIBC_2.0
```

2.4 ”%*c*”

A printf call with the format string ”%*c*” [line 4688-4696] is converted to a *putchar*() call.

```
printf("%c", 'A'); // converted to putchar('A');
```

```
#include <stdio.h>
int main(int argc, char *argv[]) {
  printf("%c", 'A');
  return 0;
}
```

Figure 5: example5.c

```
> gcc-3.4.4 -O1 example5.c
> nm --undefined-only a.out
         U __libc_start_main@@GLIBC_2.0
         U putchar@@GLIBC_2.0
```

2.5 No ′%′ in format string

No optimization if one or more ′%′ are detected in the format string [line 4699-4701].

```
#include <stdio.h>
int main(int argc, char *argv[]) {
  printf("hello%%world\n");
  return 0;
}
```

Figure 6: example6.c

```
> gcc-3.4.4 -O1 example6.c
> nm --undefined-only a.out
         U __libc_start_main@@GLIBC_2.0
         U printf@@GLIBC_2.0
```

2.6 The empty format string

Omit printf call if format string is empty [line 4706-4708].

```
printf(""); // converted to empty statement
```

```
> gcc-3.4.4 -O1 example7.c
> nm --undefined-only a.out
         U __libc_start_main@@GLIBC_2.0
```

```
#include <stdio.h>
int main(int argc, char *argv[]) {
  printf("");
  return 0;
}
```

Figure 7: example7.c

2.7 The one character string

A printf call with a format string of length one [line 4709-4718] is converted to *putchar*() call.

```
printf("A"); // converted to putchar('A');
```

```
#include <stdio.h>
int main(int argc, char *argv[]) {
  printf("A");
  return 0;
}
```

Figure 8: example8.c

```
> gcc-3.4.4 -O1 example8.c
> nm --undefined-only a.out
       U __libc_start_main@@GLIBC_2.0
       U putchar@@GLIBC_2.0
```

2.8 The string not ending with $'\backslash n'$

No optimization if the string is not ending with $'\backslash n'$ [line 4721-4739].

```
#include <stdio.h>
int main(int argc, char *argv[]) {
  printf("hello world");
  return 0;
}
```

Figure 9: example9.c

```
> gcc-3.4.4 -O1 example9.c
> nm --undefined-only a.out
       U __libc_start_main@@GLIBC_2.0
       U printf@@GLIBC_2.0
```

2.9 The string ending with $'\backslash n'$

A printf call with a simple format string ending with $'\backslash n'$ [line 4721-4739] is converted to a $puts()$ call.

```
printf("hello world\n"); // converted to puts("hello world");
```

```
#include <stdio.h>
int main(int argc, char *argv[]) {
  printf("hello world\n");
  return 0;
}
```

Figure 10: example10.c

```
> gcc-3.4.4 -O1 example10.c
> nm --undefined-only a.out
         U __libc_start_main@@GLIBC_2.0
         U puts@@GLIBC_2.0
```

3 Fine differences

3.1 %s and NULL argument

Example taken from [BUG]. The glibc [GLIa] printf implementation has the nice feature to print '(null)' in case of the format string $'\%s'$ and a pointer to NULL is given as argument.

```
#include <stdio.h>
int main(int argc, char *argv[]) {
  char *p = NULL;
  printf("%s\n", p);
  return 0;
}
```

Figure 11: bug_15685.c

```
> gcc-3.3.6 -Wall -O0 bug_15685.c
./a.out
(null)

> gcc-3.3.6 -Wall -O1 bug_15685.c
./a.out
Segmentation fault
```

As stated in the bug report by Andrew Pinski this is not a bug, because both behaviors are allowed by the standard. If the standard states *undefined behavior* the compiler and/or library is free in what to implement. You should not depend on the one or the other behavior in a portable program.

From the glibc manual: "If you accidentally pass a null pointer as the argument for a %s conversion, the GNU library prints it as (null). We think this is more useful than crashing. But it's not good practice to pass a null argument intentionally." [GLIb].

4 Comparison of GCC versions

In figure 12 note that the GCC-4.0.2 is doing this optimization even if *-O0* (no optimization) is given on the command line.

compiler	-O0	-O1	-O2	-O3
GCC-3.3.6	printf	puts	puts	puts
GCC-3.4.4	printf	puts	puts	puts
GCC-4.0.2	puts	puts	puts	puts

Figure 12: Results for *example1.c*

To get the original $printf$ call with GCC-4.0.2 you must provide $-fno-builtin$ or $-fno-builtin-printf$ as command line argument.

```
> gcc-4.0.2 -fno-builtin -O0 example1.c
> nm --undefined-only a.out
         w __gmon_start__
         w _Jv_RegisterClasses
         U __libc_start_main@@GLIBC_2.0
         U printf@@GLIBC_2.0

> gcc-4.0.2 -fno-builtin-printf -O0 example1.c
> nm --undefined-only a.out
         w __gmon_start__
         w _Jv_RegisterClasses
         U __libc_start_main@@GLIBC_2.0
         U printf@@GLIBC_2.0
```

5 Performance advantage

Some simple measurement[2] for the hello world example.

[2] On AMD Athlon(TM) XP 1800+ system

```c
#include <stdio.h>
int main(int argc, char *argv[]) {
  int i;
  for(i = 0; i < 10000000; i++) {
    printf("hello world\n");
  }
  return 0;
}
```

Figure 13: loop_printf.c

```
> gcc-3.3.6 -Wall -O0 loop_printf.c
> time ./a.out > /dev/null
real    0m2.197s
user    0m2.186s
sys     0m0.010s
> gcc-3.3.6 -Wall -O1 loop_printf.c
> time ./a.out > /dev/null
real    0m1.007s
user    0m0.997s
sys     0m0.009s
```

In figure 14 we see that the *puts* loop consumes roughly half the time of the *printf* loop.

	GCC-3.3.6	GCC-3.4.4	GCC-4.0.2
-O0	real 0m2.197s user 0m2.186s sys 0m0.010s	real 0m2.164s user 0m2.147s sys 0m0.015s	real 0m1.026s user 0m1.012s sys 0m0.014s
-O1	real 0m1.007s user 0m0.997s sys 0m0.009s	real 0m1.006s user 0m0.996s sys 0m0.011s	real 0m1.007s user 0m0.998s sys 0m0.010s

Figure 14: Times for *loop_printf.c*

6 Future work

Document the remaining optimizations mentioned in *gcc/builtins.c*. Explore advanced optimization like suggested in [GCCb].

References

[BUG] http://gcc.gnu.org/bugzilla/show_bug.cgi?id=15685.

[GCCa] http://gcc.gnu.org.

[GCCb] http://gcc.gnu.org/ml/gcc-patches/1999-01/msg00187.html.

[GLIa] http://www.gnu.org/software/libc/libc.html.

[GLIb] http://www.gnu.org/software/libc/manual/html_mono/libc.html.gz#Other%20Output%20Conve

7 Appendix

7.1 Source code *gcc-3.4.4/gcc/builtins.c*

```c
/* Expand a call to printf or printf_unlocked with argument list ARGLIST.
   Return 0 if a normal call should be emitted rather than transforming
   the function inline.  If convenient, the result should be placed in
   TARGET with mode MODE.  UNLOCKED indicates this is a printf_unlocked
   call.  */
static rtx
expand_builtin_printf (tree arglist, rtx target, enum machine_mode mode,
                       bool unlocked)
{
  tree fn_putchar = unlocked
                    ? implicit_built_in_decls[BUILT_IN_PUTCHAR_UNLOCKED]
                    : implicit_built_in_decls[BUILT_IN_PUTCHAR];
  tree fn_puts = unlocked ? implicit_built_in_decls[BUILT_IN_PUTS_UNLOCKED]
                          : implicit_built_in_decls[BUILT_IN_PUTS];
  const char *fmt_str;
  tree fn, fmt, arg;

  /* If the return value is used, don't do the transformation.  */
  if (target != const0_rtx)
    return 0;

  /* Verify the required arguments in the original call.  */
  if (! arglist)
    return 0;
  fmt = TREE_VALUE (arglist);
  if (TREE_CODE (TREE_TYPE (fmt)) != POINTER_TYPE)
    return 0;
  arglist = TREE_CHAIN (arglist);

  /* Check whether the format is a literal string constant.  */
  fmt_str = c_getstr (fmt);
  if (fmt_str == NULL)
    return 0;

  /* If the format specifier was "%s\n", call __builtin_puts(arg).  */
  if (strcmp (fmt_str, "%s\n") == 0)
    {
      if (! arglist
          || TREE_CODE (TREE_TYPE (TREE_VALUE (arglist))) != POINTER_TYPE
          || TREE_CHAIN (arglist))
        return 0;
      fn = fn_puts;
```

```
      }
    /* If the format specifier was "%c", call __builtin_putchar(arg).  */
    else if (strcmp (fmt_str, "%c") == 0)
      {
        if (! arglist
            || TREE_CODE (TREE_TYPE (TREE_VALUE (arglist))) != INTEGER_TYPE
            || TREE_CHAIN (arglist))
          return 0;
        fn = fn_putchar;
      }
    else
      {
        /* We can't handle anything else with % args or %% ... yet.  */
        if (strchr (fmt_str, '%'))
          return 0;

        if (arglist)
          return 0;

        /* If the format specifier was "", printf does nothing.  */
        if (fmt_str[0] == '\0')
          return const0_rtx;
        /* If the format specifier has length of 1, call putchar.  */
        if (fmt_str[1] == '\0')
          {
            /* Given printf("c"), (where c is any one character,)
               convert "c"[0] to an int and pass that to the replacement
               function.  */
            arg = build_int_2 (fmt_str[0], 0);
            arglist = build_tree_list (NULL_TREE, arg);
            fn = fn_putchar;
          }
        else
          {
            /* If the format specifier was "string\n", call puts("string").  */
            size_t len = strlen (fmt_str);
            if (fmt_str[len - 1] == '\n')
              {
                /* Create a NUL-terminated string that's one char shorter
                   than the original, stripping off the trailing '\n'.  */
                char *newstr = (char *) alloca (len);
                memcpy (newstr, fmt_str, len - 1);
                newstr[len - 1] = 0;

                arg = build_string_literal (len, newstr);
                arglist = build_tree_list (NULL_TREE, arg);
```

```
                fn = fn_puts;
              }
            else
              /* We'd like to arrange to call fputs(string,stdout) here,
                 but we need stdout and don't have a way to get it yet.  */
              return 0;
          }
      }

  if (!fn)
    return 0;
  return expand_expr (build_function_call_expr (fn, arglist),
                      target, mode, EXPAND_NORMAL);
}
```