GCC hacks in the Linux kernel

Discover GCC extensions for the C language

M. Tim Jones

November 18, 2008

The Linux® kernel uses several special capabilities of the GNU Compiler Collection (GCC) suite. These capabilities range from giving you shortcuts and simplifications to providing the compiler with hints for optimization. Discover some of these special GCC features and learn how to use them in the Linux kernel.

GCC and Linux are a great pair. Although they are independent pieces of software, Linux is totally dependent on GCC to enable it on new architectures. Linux further exploits features in GCC, called *extensions*, for greater functionality and optimization. This article explores many of these important extensions and shows you how they're used within the Linux kernel.

GCC in its current stable version (version 4.3.2) supports three versions of the C standard:

- The original International Organization for Standardization (ISO) standard of the C language (ISO C89 or C90)
- ISO C90 with amendment 1
- The current ISO C99 (the default standard that GCC uses and that this article assumes)

Note: This article assumes that you are using the ISO C99 standard. If you specify a standard older than the ISO C99 version, some of the extensions described in this article may be disabled. To specify the actual standard that GCC uses, you can use the `-std` option from the command line. Use the GCC manual to verify which extensions are supported in which versions of the standard (see Related topics for a link).

Applicable versions

This article focuses on the use of GCC extensions in the 2.6.27.1 Linux kernel and version 4.3.2 of GCC. Each C extension refers to the file in the Linux kernel source where the example can be found.

The available C extensions can be classified in several ways. This article puts them in two broad categories:

- *Functionality* extensions bring new capabilities from GCC.

- *Optimization* extensions help you generate more efficient code.

Functionality extensions

Let's start by exploring some of the GCC tricks that extend the standard C language.

Type discovery

GCC permits the identification of a type through the reference to a variable. This kind of operation permits a form of what's commonly referred to as *generic programming*. Similar functionality can be found in many modern programming languages such as C++, Ada, and the Java™ language. Linux uses `typeof` to build type-dependent operations such as `min` and `max`. Listing 1 shows how you can use `typeof` to build a generic macro (from ./linux/include/linux/kernel.h).

Listing 1. Using `typeof` to build a generic macro

```
#define min(x, y) ({    \
 typeof(x) _min1 = (x);   \
 typeof(y) _min2 = (y);   \
 (void) (&_min1 == &_min2);  \
 _min1 < _min2 ? _min1 : _min2; })
```

Range extension

GCC includes support for ranges, which can be put to use in many areas of the C language. One of those areas is on `case` statements within `switch`/`case` blocks. In complex conditional structures, you might typically depend on cascades of `if` statements to achieve the same result that is represented more elegantly in Listing 2 (from ./linux/drivers/scsi/sd.c). The use of `switch`/`case` also enables compiler optimization by using a jump table implementation.

Listing 2. Using ranges within `case` statements

```
static int sd_major(int major_idx)
{
 switch (major_idx) {
 case 0:
  return SCSI_DISK0_MAJOR;
 case 1 ... 7:
  return SCSI_DISK1_MAJOR + major_idx - 1;
 case 8 ... 15:
  return SCSI_DISK8_MAJOR + major_idx - 8;
 default:
  BUG();
  return 0; /* shut up gcc */
 }
}
```

Ranges can also be used for initialization, as shown below (from ./linux/arch/cris/arch-v32/kernel/smp.c). In this example, an array is created of `spinlock_t` with a size of `LOCK_COUNT`. Each element of the array is initialized with the value `SPIN_LOCK_UNLOCKED`.

```
/* Vector of locks used for various atomic operations */
spinlock_t cris_atomic_locks[] = { [0 ... LOCK_COUNT - 1] = SPIN_LOCK_UNLOCKED};
```

Ranges also support more complex initializations. For example, the following code specifies initial values for sub-ranges of an array.

```
int widths[] = { [0 ... 9] = 1, [10 ... 99] = 2, [100] = 3 };
```

Zero-length arrays

In standard C, at least one element of an array must be defined. This requirement tends to complicate code design. However, GCC supports the concept of zero-length arrays, which can be particularly useful for structure definitions. This concept is similar to the flexible array member in ISO C99, but it uses a different syntax.

The following example declares an array with zero members at the end of a structure (from ./linux/drivers/ieee1394/raw1394-private.h). This allows the element in the structure to reference memory that follows and is contiguous with the structure instance. You may find this useful in cases where you need to have a variable number of array members.

```
struct iso_block_store {
        atomic_t refcount;
        size_t data_size;
        quadlet_t data[0];
};
```

Determining call address

In many instances, you may find it useful or necessary to determine the caller of a given function. GCC provides the built-in function `__builtin_return_address` for just this purpose. This function is commonly used for debugging, but it has many other uses within the kernel.

As shown in the code below, `__builtin_return_address` takes an argument called `level`. The argument defines the level of the call stack for which you want to obtain the return address. For example, if you specify a `level` of `0`, you are requesting the return address of the current function. If you specify a `level` of `1`, you are requesting the return address of the calling function (and so on).

```
void * __builtin_return_address( unsigned int level );
```

The `local_bh_disable` function in the following example (from ./linux/kernel/softirq.c) disables soft interrupts on the local processor to prevent softirqs, tasklets, and bottom halves from running on the current processor. The return address is captured using `__builtin_return_address` so that it can be used for later tracing purposes.

```
void local_bh_disable(void)
{
        __local_bh_disable((unsigned long)__builtin_return_address(0));
}
```

Constant detection

GCC provides a built-in function that you can use to determine whether a value is a constant at compile-time. This is valuable information because you can construct expressions that can be optimized through constant folding. The `__builtin_constant_p` function is used to test for constants.

The prototype for `__builtin_constant_p` is shown below. Note that `__builtin_constant_p` cannot verify all constants, because some are not easily proven by GCC.

```
int __builtin_constant_p( exp )
```

Linux uses constant detection quite frequently. In the example shown in Listing 3 (from ./linux/include/linux/log2.h), constant detection is used to optimize the `roundup_pow_of_two` macro. If the expression can be verified as a constant, then a constant expression (which is available for optimization) is used. Otherwise, if the expression is not a constant, another macro function is called to round up the value to a power of two.

Listing 3. Constant detection to optimize a macro function

```
#define roundup_pow_of_two(n)   \
(       \
 __builtin_constant_p(n) ? (  \
  (n == 1) ? 1 :   \
  (1UL << (ilog2((n) - 1) + 1)) \
       ) :  \
 __roundup_pow_of_two(n)   \
)
```

Function attributes

GCC provides a variety of function-level attributes that allow you to provide more data to the compiler to assist in the optimization process. This section describes some of these attributes that are associated with functionality. The next section describes attributes that affect optimization.

As shown in Listing 4, the attributes are aliased by other symbolic definitions. You can use this as a guide to help read the source references that demonstrate the use of the attributes (as defined in ./linux/include/linux/compiler-gcc3.h).

Listing 4. Function attribute definitions

```
# define __inline__     __inline__      __attribute__((always_inline))
# define __deprecated           __attribute__((deprecated))
# define __attribute_used__     __attribute__((__used__))
# define __attribute_const__     __attribute__((__const__))
# define __must_check            __attribute__((warn_unused_result))
```

The definitions shown in Listing 4 reflect some of the function attributes available in GCC. They are also some of the most useful function attributes in the Linux kernel. Following are explanations of how you can best use these attributes:

- `always_inline` tells GCC to inline the specified function regardless of whether optimization is enabled.
- `deprecated` tells you when a function has been deprecated and should no longer be used. If you attempt to use a deprecated function, you receive a warning. You can also apply this

attribute to types and variables to encourage developers to wean themselves from those kernel assets.

- `__used__` tells the compiler that this function is used regardless of whether GCC finds instances of calls to the function. This can be useful in cases where C functions are called from assembly.
- `__const__` tells the compiler that a particular function has no state (that is, it uses the arguments passed in to generate a result to return).
- `warn_unused_result` forces the compiler to check that all callers check the result of the function. This ensures that callers are properly validating the function result so that they can handle the appropriate errors.

Following are examples of these function being used in the Linux kernel. The `deprecated` example comes from the architecture non-specific kernel (./linux/kernel/resource.c), and the `const` example comes from the IA64 kernel source (./linux/arch/ia64/kernel/unwind.c).

```
int __deprecated __check_region(struct resource
    *parent, unsigned long start, unsigned long n)

static enum unw_register_index __attribute_const__
    decode_abreg(unsigned char abreg, int memory)
```

Optimization extensions

Now, let's explore some of the GCC tricks available to produce the best machine code possible.

Branch prediction hints

One of the most ubiquitous optimization techniques used in the Linux kernel is `__builtin_expect`. When working with conditional code, you often know which branch is most likely and which is not. If the compiler has this prediction information, it can generate the most optimal code around the branch.

As shown below, use of `__builtin_expect` is based on two macros called `likely` and `unlikely` (from ./linux/include/linux/compiler.h).

```
#define likely(x) __builtin_expect(!!(x), 1)
#define unlikely(x) __builtin_expect(!!(x), 0)
```

With `__builtin_expect`, the compiler can make instruction-selection decisions that favor the prediction information you provide. This keeps the code most likely to execute close to the condition. It also improves caching and instruction pipelining.

For example, if the conditional is marked "likely" then the compiler can place the True portion of the code immediately following the branch (which will not be taken). The False portion of the conditional would then be available through the branch instruction, which is less optimal but also less likely. In this way, the code is optimized for the most likely case.

Listing 5 shows a function that uses both the `likely` and `unlikely` macros (from ./linux/net/core/datagram.c). The function expects that the `sum` variable will be zero (`checksum` is valid for the packet) and that the `ip_summed` variable is not equal to `CHECKSUM_HW`.

Listing 5. Example use of the likely and unlikely macros

```
unsigned int __skb_checksum_complete(struct sk_buff *skb)
{
        unsigned int sum;

        sum = (u16)csum_fold(skb_checksum(skb, 0, skb->len, skb->csum));
        if (likely(!sum)) {
                if (unlikely(skb->ip_summed == CHECKSUM_HW))
                        netdev_rx_csum_fault(skb->dev);
                skb->ip_summed = CHECKSUM_UNNECESSARY;
        }
        return sum;
}
```

Prefetching

Another important method of improving performance is through caching of necessary data close to the processor. Caching minimizes the amount of time it takes to access the data. Most modern processors have three classes of memory:

- Level 1 cache commonly supports single-cycle access
- Level 2 cache supports two-cycle access
- System memory supports longer access times

To to minimize access latency, and thus improve performance, it's best to have your data in the closest memory. Performing this task manually is called *prefetching*. GCC supports manual prefetching of data through a built-in function called `__builtin_prefetch`. You use this function to pull data into the cache shortly before it's needed. As shown below, the `__builtin_prefetch` function takes three arguments:

- The address of the data
- The `rw` parameter, which you use to indicate whether the data is being pulled in for Read or preparing for a Write operation
- The `locality` parameter, which you use to define whether the data should be left in cache or purged after use

```
void __builtin_prefetch( const void *addr, int rw, int locality );
```

Prefetching is used extensively by the Linux kernel. Most often it is used through macros and wrapper functions. Listing 6 is an example of a helper function that uses a wrapper over the built-in function (from ./linux/include/linux/prefetch.h). The function implements a preemptive look-ahead mechanism for streamed operations. Using this function can generally result in better performance by minimizing cache misses and stalls.

Listing 6. Wrapper function for range prefetching

```
#ifndef ARCH_HAS_PREFETCH
#define prefetch(x) __builtin_prefetch(x)
#endif

static inline void prefetch_range(void *addr, size_t len)
{
#ifdef ARCH_HAS_PREFETCH
 char *cp;
 char *end = addr + len;

 for (cp = addr; cp < end; cp += PREFETCH_STRIDE)
  prefetch(cp);
#endif
}
```

Variable attributes

In addition to the function attributes discussed earlier in this article, GCC provides attributes for variables and type definitions. One of the most important of these is the `aligned` attribute, which is used for object alignment in memory. In addition to being important for performance, object alignment may be required for particular devices or hardware configurations. The `aligned` attribute takes a single argument that specifies the desired type of alignment.

The following example is used for software suspend (from ./linux/arch/i386/mm/init.c). The `PAGE_SIZE` object is defined as needing page alignment.

```
char __nosavedata swsusp_pg_dir[PAGE_SIZE]
 __attribute__ ((aligned (PAGE_SIZE)));
```

The example in Listing 7 illustrates a couple of points regarding optimization:

- The `packed` attribute packs the elements of a structure so that it consumes the least amount of space possible. This means that if a `char` variable is defined, it will consume no more than a byte (8 bits). Bit fields are compressed into a bit rather than consuming more storage.
- This source presentation is optimized by use of a single `__attribute__` specification that defines multiple attributes with a comma-delimited list.

Listing 7. Structure packing and setting multiple attributes

```
static struct swsusp_header {
        char reserved[PAGE_SIZE - 20 - sizeof(swp_entry_t)];
        swp_entry_t image;
        char    orig_sig[10];
        char    sig[10];
} __attribute__((packed, aligned(PAGE_SIZE))) swsusp_header;
```

Going further

This article provides only a glimpse of the techniques made available by GCC in the Linux kernel. You can read more about all the available extensions for both C and C++ in the GNU GCC manual (see Related topics for a link). And although the Linux kernel makes great use of these extensions, they are all available to you for use in your own applications as well. As GCC continues to evolve, new extension are sure to further improve the performance and increase the functionality of the Linux kernel.

Related topics

- The GNU Compiler Collection is the source for all things GCC. Here, you'll find news and the latest source for GCC (including historical and leading-edge distributions). You can also find the detailed histories for each release as well as the online documentation, which provides a detailed treatment of GCC and all extensions.
- Experimental GCC extensions is a GNU-maintained list of those extensions that are not yet in the standard distribution. This is a great place to check for upcoming changes to GCC extensions.
- In the article "Get to know GCC 4" (developerWorks, October 2008), learn more about the various changes in the fourth major release of GCC, called GCC4. This article introduces you to GCC4 and its evolution over the last four minor releases.
- In the developerWorks Linux zone, find more resources for Linux developers, including developers who are new to Linux.
- See all Linux tips and Linux tutorials on developerWorks.