Effective C With The GCC And GLIBC

“long long long is too long for GCC”

Hagen Paul Pfeifer
hagen@jauu.net

http://protocol-laboratories.net

3rd June 2007

What the heck ...

- Today we talk about advanced GCC and GLIBC functionality, but ...
  - ... not in a sense of pure academic research (compiler constructions, whatever)
  - Intention is to improve coding skills with well known and often less known techniques
  - At the end: a GCC/GLIBC outlook are envisaged to wake up your hacker capabilities
- Anyway: like in all other areas; if your work depends on a heavy utilization of your compiler suite and the standard library, then <u>invest time</u> to study GCC and GLIBC.
- So lets get started!

Agenda

- GCC - GNU Compiler Collection
- GLIBC - GNU C Library

Chapter 1

GNU Compiler Collection

Use const

- Concept of “something is not modifiable” by variable declaration
- `const uint32_t *ptr` → pointer to `const uint32_t`
- `uint8_t *const ptr` → `const` pointer to `uint32_t`
- Be warned: modify `const` declared values through pointers is valid (undefined behaviour, see `const` as a MAY, not MUST be immutable)
- Allow compiler to store value in a non-modifiable section
- Additional: the compiler can do some consistency checks
- FYI: think about a system where there is no real memory protection – how/why should a real low level programming standard prevent `const` memory changes? That is the answer – C **is** a low-level programming standard!

USE .rodata

- `char *msg = "Whatever, Wherever";` (global declared)
- Some updates/improvements desired!
- Programming Subsidence Slope:
  1. Variable `msg` not needed
  2. Stored in `.data` segment
  3. Relocation needed
- `const char msg[] = "Whatever, Wherever";` (inside scope, Stack)
  1. Allocate Memory on stack and copy string to it

Use `strlen()`

- Partly the compiler can calculate the result at compile time
- Cache the result if re-use it again
- PowerPC 4xx: `dlmzb` (determine left-most zero byte) → `-O2 -mcpu=440`

Avoid type casts

- Avoid type casts whenever possible (especially pointer casts)
  - They usually hide errors (disables type checking)
  - Variable access is based on type of variable - not the cast
  - Often dangerous and very uncontrolled
  - Don't shut up compiler warnings with casts!
  - ISO C automatically converts `void *` when necessary
  - This doesn't happened on traditional compiler
- `float *fp = (float *) ip;` (ip defined as `int *`)
  - Undefined behavior (C Standard Document)
  - `sizeof(float)` vs. `sizeof(int)`
  - Older compiler interpret `ip` as a float
  - Newer ones doesn't do that! (Uninitialized value or zero)

- The real cause why a compiler check this is the rearrangement of code (it is not primarily for the user (c had no exceptions ;-) it is for code optimization purpose)
- Tip: if you really want to interpret values as values of other types then use `unions`

Function Inlining

- Understand What The Compiler Will Generate And See The Overall Context!
- Inlining isn't a make code faster, securer, cuter, whatever flag at all
- `__attribute__((always_inline));`, `-finline-functions`, `-Winline`
- Type checking at all - compared to macros
- Use `-fno-inline` if you want to debug your code

Code Optimization

- Optimize the excepted case (`gcov`)
- `vi gcc/toplev.c +/optimize` (understand[tm] optimization flags)
- `-march=ARCH` (gcc 4 introduce `-march=native` – this utilize `CPUID` instruction at compile time)
- `-msse` generate code for built in functions (e.g. `(gcc/config/i386/i386.c`)

```
#ifndef __cacheline_aligned
#define __cacheline_aligned                                    \
   __attribute__((__aligned__(SMP_CACHE_BYTES),    \
          __section__(".data.cacheline_aligned")))
#endif /* __cacheline_aligned */

#define __read_mostly __attribute__((__section__(".data.read_mostly")))
```

- pahole (`/pub/scm/linux/kernel/git/acme/pahole.git`) (`oops.ghostprotocols.net:81/blog`)

VLA - Variable Length Arrays

- C99 Standard or/and GCC extension
- It is really fast and waster nearly no space
- `alloca()` is function local - Not scope local (`brace level`)
- Disadvantages: no clean error messages if you request to much memory
- Example: (onlinedocs/gcc 5.14)

```
FILE *
concat_fopen (char *s1, char *s2, char *mode)
{
  char str[strlen (s1) + strlen (s2) + 1];
  strcpy (str, s1);
  strcat (str, s2);
  return fopen (str, mode);
}
```

- parameter forward declaration (GNU extension, no ISO C99):

```
struct entry
tester (int len; char data[len][len], int len)
{ /* ... */ }
```

__section__

- `readelf -S elf-file`
- Kernel Section Example:
  - Naturally: all writeable (`!const`) data are located in section `.data`:
    - Data frequently but rarely written causes needlessly cache misses
    - Data are oft written once (e.g. at module start-up)
    - Often changed data are awkward on SMP system (Cache Consistency, MESI)
    - Approach: save less frequently touched data in a another location so that this (mostly readonly) cacheline mustn't reloaded all the time
  - `#define __read_mostly __attribute__((__section__(".data.read_mostly")))`
    - prevent cache line pollution (read from often and rarely written variables)
    - False sharing, Cache Coherence, MESI

Avoid False Sharing

- Remember: not only obviously shared data between threads is affected – any data that is on the same cache line is also affected (false sharing)
- Background: if a processor modify a cache line it “broadcast” this event to all other processors and they invalidate this cache line
- In the case of two - often accessed variables - are on one cache line, this can lead to tremendous effects!
- Cache line is atomic (for invalidation tagging)
- threaded application
- Thread A write to cache line 1; this cache line gets now invalidated to the other thread; cache miss for thread B; Memory access
- Global arrays are a common example: `int sum[THREAD_NO]`
- Way out:
  - Pad data element (each element lie on separate cache line)

- local stack copy

Avoid False Sharing

- Therefore: all synchronisation variables on a own cache line and no other data on the line
- How big is the cache line on my CPU? → `CPUID` (P3: 32bytes; P4: 128bytes (sub divided into 64byte chunks))
- Intel Example (lightly modified version ;-):

```
#define CACHE_LINE_SIZE 128
struct syn_str { int s_variable; };
void *p = malloc(sizeof(struct syn_str) + (CACHE_LINE_SIZE - 1));
syn_str *align_p = (syn_str *)((((int) p) + (CACHE_LINE_SIZE - 1)) & - CACHE_LINE_SIZE);
#undef CACHE_LINE_SIZE
```

- Superiorly: icc: `_declspec(align(128))`, gcc: `__attribute__ ((aligned(32)))`

Avoid False Sharing

- `include/linux/mmzone.h`:

```
/*
 * zone->lock and zone->lru_lock are two of the hottest locks in the kernel.
 * So add a wild amount of padding here to ensure that they fall into separate
 * cachelines.  There are very few zone structures in the machine, so space
 * consumption is not a concern here.
 */
#if defined(CONFIG_SMP)
struct zone_padding {
      char x[0];
} ____cacheline_internodealigned_in_smp;
#define ZONE_PADDING(name)    struct zone_padding name;
#else
#define ZONE_PADDING(name)
#endif

#define ____cacheline_internodealigned_in_smp  \
  __attribute__((__aligned__(1 << (INTERNODE_CACHE_SHIFT))))
```

- `INTERNODE_CACHE_SHIFT`:
  - “The maximum alignment needed for some critical structures. These could be inter-node cacheline sizes/L3 cacheline size etc. Define this in

asm/cache.h for your arch” (`linux/cache.h`)

- x86 | ia64 : `CONFIG_X86_L1_CACHE_SHIFT` (5 (32), . . .)
- Alpha: 6 (64)
- Powerpc: 4, 5, 7 (32, 64, 128)
- s390: 8 (512)

Various

- Should be obvious, but: a integer isn't always 4 byte wide (`{u}intN_t`, `...stdint.h` (ISO C99: 7.18 Integer types))
- `{U}INTn_MAX`
- `size_t`
  - `size_t` unsigned integer which is able to represent the size of an object
  - Result of `sizeof()` will always fit into `size_t`
  - Limit: `SIZE_MAX`
- Align Data Structures on Cache Boundaries
- `-minline-all-stringops`
- `-march=native`
  - `gcc/config/i386/driver-i386.c:host_detect_local_cpu()`
  - `L1_ASSOC` associative cache

- L1_SIZEKB
- L1_LINE

- Over/Underflow
  - `int i=0;while(i >= 0) {i++; /* something */ }`
  - C Standard: Undefined Behavior (no wrapping, . . . , nothing)
  - GCC 4.3: `-Wstrtict-overflow={1,2,3,4,5}`
- GCC 4.4 (maybe later)
  - Inlining for object files (inlining in linking phase, intermediate representation code also into object file; inlining betwwen two object files (e.g. libraries))
  - Whole programm optimization - not only for object file chunks
  - LTO object (Link time object)

Additional

- How is x typedefed/defined (e.g. `suseconds_t`)? (or how to handle several levels of indirection for macros?)
  - GCC tip: `gcc -E suseconds_t.c -o - | grep suseconds_t -`
  - Vim tip: `[I` (often faster but `gcc -E` approach is safer)
- Subversion Hook:
  - Use GCC to check syntax of source code: `gcc -fsyntax-only *.c`
- `-ftrapv`: “This option generates traps for signed overflow on addition, subtraction, multiplication operations”
- Floating point trapping
  - feenableexcept(3) → control the behaviour of individual exceptions
- `-fmudflap -lmudflap`

Chapter 2

GNU C Library

Know Your GLIBC (and implementation of their functions!)

- Even if the GLIBC development reminds to closed source . . . ;-(
- Simple example: `fputs()` versus `printf()` versus `write()`
- `posix_memalign()` `sysconf(_SC_PAGESIZE)`
- Some sweetmeats (ok, some are broken by design an superfluous):
  - `epoll()`, `futex()`, `regex (regcomp(), regexec(), ...)`,
  - `glob()`, `posix_fallocate()`, `posix_fadvise()`, `backtrace()`
  - `writev()`, `sync_file_range()`, `msync`
  - `__fbufsize`, `__fpending`, `__fsetlocking`
  - `strfry()`, `memfrob()`, `l64a()`, `hcreate()`, `backtrace()`
  - `getsubopt()`, `lfind()`, `tsearch()`

- `dprintf(int fd, const char *format, ...);`

Memory

- `malloc()`ed memory is guaranteed aligned (8byte): therefore it can hold any type of data and this memory is cache aware aligned for most cases. (16byte boundary for 64bit architecutres)
- If you need higher alignment wrote your own function or use `posix_memalign()`
- If you are lazy: write a malloc wrapper: e.g. xmalloc()
- `malloc()` tunning: `mallopt()`
- KS Tunning:
  - `overcommit_memory` $0, 1, 2$
  - FYI: until pages are touched, real assigned take place (implement your own malloc (brk(), mmap() and allocate mind-boggling amount of memory)
- If all fails: `mm/oom_kill.c` ;)

GLIBC Memory Giveaways

- *** glibc detected *** nmap: malloc(): memory corruption: 0x0f718a50 ***+
- “How can I disable this message?”
- There are nearly NO false positive - please do not ignore it
- Tip: use `valgrind --tool=memcheck a.out` to find the error
- `MALLOC_CHECK_` $= 0, 1, 2$

USE glibc at all!

- If you operate on memory: use mem*; if you operate on null terminated arrays: use str*
- If you know the size of an array: use mem*, memorize it and don’t recalculate this values again and again

Fin – Last but not least

- Pay attention to (unconditional) branches, reorder your code (higher instruction cache miss ratio)
- If your code should/must be portable, avoid `some` gcc/glibs hacks (ignore this if you like `#ifdef/#endif` wasting ;-)
- At least: keep the overall program context in mind (skill-level of developers, hot-spots of program, execution context, . . .)
- At the end: use optimal data structures and algorithm and your are a winner! ;-)
- Questions?

Additional Information

- Links:
  - The GNU C Library
  - SSE4 Introduction
  - How to Align Data Structures on Cache Boundaries
- Books/Papers (without links)
  - AP-949 Using Spin-Loops on Intel Pentium 4 Processor and Intel Xeon Processor
  - Fast Synchronisation for Chip Multiprocessors (really nice approach for synchronisation mechanism on chip multi processors)
  - Architectural Analysis and Instruction-Set Optimization for Design of Network Protocol Processors (they study the TCP/IP stack with SimpleScalarTool and change cache attributes to see performance effects - increase instruction cache size, increase set associativity, increase line

size)

- Network Algorithmics – An Interdisciplinary approach to designing fast networked devices
- Unix Systems for Modern Architectures, Symmetric Multiprocessing and Caching for Kernel Programmers

Contact

- Hagen Paul Pfeifer
- EMail: hagen@jauu.net
  - Key-ID: `0x98350C22`
  - Fingerprint: `490F 557B 6C48 6D7E 5706 2EA2 4A22 8D45 9835 0C22`

Document-ID: 3578d4b9d9e28c56787c103ab2876629bc70a089

Branch Optimization

- Reorder Code:

```
if (false_usually) {
        if (true_usually) {
        }
}

if (false_usually && true_usually) {
}

if (true_usually || false_usually) {
}
```