Aliasing and endianness in C99/C++11

and

data transfer between hard real-time systems on modern RISC processors

Erik Alapää

FOSS-Sthlms Linuxhackardag 1:a juni 2013

Background

Issues such as aliasing and endianness are important in many contexts, such as networking.

This talk was inspired by work on a messaging protocol between two RISC CPUs, one (here called system B) running a common hard real-time OS, the other (system S) running without OS, and almost all code running in interrupt context.

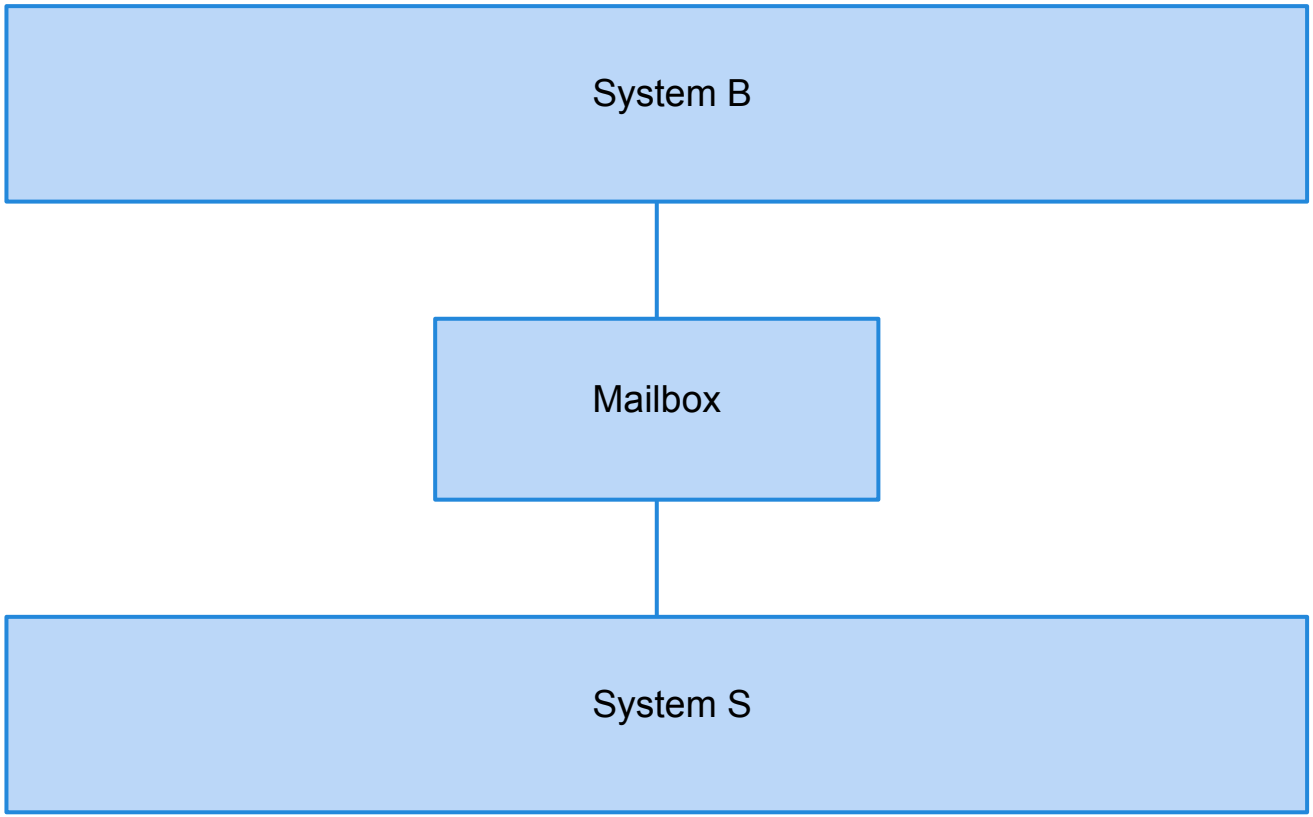
Background, continued

System B (B for Big): Big-endian, ~2 GHz PowerPC CPU, 64 MB RAM, Full-blown hard real-time OS.

System S (S for Small): Little-endian, ~100 MHz RISC CPU (2 different architectures supported), 64K data RAM, 128K RAM for instructions (!) No MMU (no virtual memory), no heap, no OS. So no malloc/free, new/delete. No C++ STL, except for in tests.

Both systems are built with gcc, mainly C++ but also some C. Messaging protocol supports both C and C++.

System overview with mailbox



Mailbox

- Architecture is word-oriented. Reading and writing 32-bit words is handled transparently with regard to endianness.
- Typical mailbox message is 3 words, one word for header, one word for message type, and one word for message data, if necessary
- Maximum mail size is 64 words, i.e. 256 bytes.
- Fragmented mails used for messages > 64 words. (4-bit frag # in header => 16 fragments => max msg size = 16x64 words)

Definition

Big-endian: Multibyte quantities are laid out in memory with the *most significant byte on the lowest byte address*.

Little-endian: Multibyte quantities are laid out in memory with the *least significant byte on the lowest byte address*.

Intel X86: little-endian. Most RISC CPUs and Internet: big-endian. ARM and PPC can do both.

Data structures, bitfields (non-portable, removed)

```
struct B2SIfHeader {
   U32 size: 12;  // 12-bit payload
                  // length in bytes
   U32 fragNr: 4; // 4-bit fragment nr
   U8 src;
   U8 dest;
};
```

The size should always occupy the 12 most significant bits in the 32-bit header, followed by fragNr, src, dest.

On little-endian machines, the order of the struct subelements is reversed, since most (all?) C compilers want the first struct subelement on the lowest byte address.

Bitfields and portability

For some reason, the C/C++ standard commitees have never made bitfields portable (does anyone in audience know why? I do not).

In mailbox, we ended up removing all bitfields and handling the message header as a complete U32 (our uint32_t) using only bitshifts and bitwise AND/OR. Note that CPU registers are usually at least 32 bits, so e.g masking out the 12 most significant bits can be done endianness-independently.

As a sidenote, we also tried removing all function-like macros and used std C/C++ inline functions instead. Macros are stone-age, IMHO.

With this background...

... now we are ready to talk about strict aliasing and the 'restrict' keyword! We also get more understanding of the difference between an array and a pointer in C/C++.

What is strict aliasing?

- assumption by the compiler that fundamentally different pointer types do not point to the same memory area, i.e. do no alias each other
- int* and short int* are fundamentally different.
- const int*, unsigned int* and int* are not fundamentally different
- a char* can alias anything, even under strict aliasing
- assymetry: char* can alias anything, but nothing can alias a char*...

What is strict aliasing?, cont.

- *types within a union may alias each other*
- strict aliasing allows the compiler to produce efficient code
- present in C and C++ long before C++97/C99, but made more visible when gcc started enforcing strict aliasing with e.g. -O3 optimization level

The 'restrict' keyword in C99

Even under strict aliasing, pointers of the same type may alias. This can force the compiler to produce extremely slow code. Next example shows a typical case.

Example

```
typedef struct vector3  vector3;

struct vector3
{
  float x;
  float y;
  float z;
};


void move(
    vector3* velocity,
    vector3* position,
    vector3* acceleration,
    float    time_step,
    size_t   count);
```

How do we know that e.g. the vector3 pointers do not alias? They may overlap...

Example, continued

If, instead, we used arrays at file scope, the compiler would know that the arrays were completely non-overlapping *data stripes:*

```
vector3 velocity[PARTICLE_COUNT];
vector3 position[PARTICLE_COUNT];
vector3 acceleration[PARTICLE_COUNT];

void
move(float time_step);
```

Example, continued

The *restrict* keyword enables us to specify the same data stripe structure without file scope arrays:

Example, continued

```
void move(
    vector3* velocity,
    vector3* position,
    vector3* acceleration,
    float    time_step,
    size_t   count,
    size_t   stride )
{
  float* restrict acceleration_x = &acceleration->x;
  float* restrict velocity_x     = &velocity->x;
  float* restrict position_x     = &position->x;
  float* restrict acceleration_y = &acceleration->y;
  float* restrict velocity_y     = &velocity->y;
  float* restrict position_y     = &position->y;
  float* restrict acceleration_z = &acceleration->z;
  float* restrict velocity_z     = &velocity->z;
  float* restrict position_z     = &position->z;

  ...
```

Example, continued

Tree of pointers. Don't use the parent pointers in same scope, this will violate the *restrict* contract and cause UB (Undefined Behavior). Might crash your program or cause Michael Bublé to start singing on your mother's doorstep...

```
                |---> velocity_x
velocity -------|---> velocity_y
                |---> velocity_z

                |---> position_x
position -------|---> position_y
                |---> position_z

                |---> acceleration_x
acceleration ---|---> acceleration_y
                |---> acceleration_z
```

Retrofit existing code with *restrict*

New official declaration of memcpy():

```
void* memcpy(
    void* restrict s1,
    const void* restrict s2,
    size_t n);
```

Reference

http://www.unix.com/apropos-man/All/0/man/

indicates that OpenSolaris 2009.06, POSIX and OSX 10.6.2 have 'restrict' memcpy, Linux and FreeBSD 8.0 do not.

Modified memcpy, more info

So, no restrict in Linux memcpy?

From /usr/include/string.h on Kubuntu 12.04.2 LTS and SuSE 10.4:

```
/* Copy N bytes of SRC to DEST.  */
extern void *memcpy (void *__restrict __dest,
     __const void *__restrict __src, size_t __n)

     __THROW __nonnull ((1, 2));
```

Want to read more?

http://cellperformance.beyond3d.com/articles/2006/05/demystifying-the-restrict-keyword.html

No 'unrestrict' in C99...

```
Weblkml.org
From (Linus Torvalds)
Subject   Re: Invalid compilation without -fno-strict-aliasing
Date Wed, 26 Feb 2003 17:26:37 +0000 (UTC)

In article <20030225234646.GB30611@bougret.hpl.hp.com>,
Jean Tourrilhes  <jt@bougret.hpl.hp.com> wrote:
>
>    It looks like a compiler bug to me...

Why do you think the kernel uses "-fno-strict-aliasing"?

The gcc people are more interested in trying to find out what can be
allowed by the c99 specs than about making things actually _work_. The
aliasing code in particular is not even worth enabling, it's just not
possible to sanely tell gcc when some things can alias.

...

I tried to get a sane way a few years ago, and the gcc developers really
didn't care about the real world in this area. I'd be surprised if that
had changed, judging by the replies I have already seen.

I'm not going to bother to fight it.

                Linus
```

Exercise

What does the following code do? What was the intent of the programmer? Put the code in a program, compile it with '-O3' (optimization level 3).

```
uint32_t
swap_words(uint32_t arg)
{
    uint16_t* const sp = (uint16_t*)&arg;
    uint16_t hi = sp[0];
    uint16_t lo = sp[1];

    sp[1] = hi;
    sp[0] = lo;

    return (arg);
}
```

Aliasing bug!

Code on previous slide does not comply with strict aliasing rules. However, testing it with gcc on a 32-bit X86 Kubuntu machine did not show the bug. Neither did gcc on Raspbian (ARM1176JZFS CPU). Same with clang, the gcc competitor. A good developer i know got the bug to show on an ARM-based cell phone with gcc.

I first heard of strict aliasing in 2006, when a program I wrote worked with -O0, but not with -O3...

So, how do we use unions to avoid aliasing bugs?

First, we define a union containing the data (*not* the pointers) we want to be able to alias:

```
typedef union
{
    uint32_t u32;
    uint16_t u16[2];
} U32;
```

Using unions to avoid aliasing bugs, continued

```
uint32_t
swap_words(uint32_t arg)
{
    U32 in;
    uint16_t lo;
    uint16_t hi;

    in.u32    = arg;
    hi        = in.u16[0];
    lo        = in.u16[1];
    in.u16[0] = lo;
    in.u16[1] = hi;

    return (in.u32);
}
```

Standards legaleze:

Type-punning through a union is not strictly allowed by the standards. However, type-punning is so common in C and C++ that it always works in practice. If it does not, your compiler is broken.

Casting through char* also works

It is always presumed that a char* may refer to an alias of any object. It is therefore quite safe, if perhaps a bit *unoptimal* (for architecture with wide loads and stores) to cast any pointer of any type to a char* type.

Might also be useful to write a memcpy with char* instead of void*. However, one advantage of your compiler's memcpy may be that it is more efficent.

Casting through char*, continued

```
uint32_t swap_words(uint32_t arg)
{
    char* const cp = (char*)&arg;
    const char  c0 = cp[0];
    const char  c1 = cp[1];
    const char  c2 = cp[2];
    const char  c3 = cp[3];
    cp[0] = c2;
    cp[1] = c3;
    cp[2] = c0;
    cp[3] = c1;
    return (arg);
}
```

Aliasing warnings in gcc

Compiler switches from open-source Enea LINX for Linux:

```
-std=gnu99 -Wall -pedantic -Wstrict-aliasing=2 -O2 -DNDEBUG
```

gcc 4.1.2 on a 32-bit X86 with SuSE Linux 10.4 does not give aliasing warnings when compiling linx_bmark from linx-2.5.1 package.

With gcc-4.7.2 on same machine we get e.g.

```
linx_bmark_attach.c: In function 'test_once':
linx_bmark_attach.c:108:23: warning: dereferencing type-punned pointer
might break strict-aliasing rules [-Wstrict-aliasing]
linx_bmark_attach.c:120:22: warning: dereferencing type-punned pointer
might break strict-aliasing rules [-Wstrict-aliasing]
linx_bmark_attach.c:131:22: warning: dereferencing type-punned pointer
might break strict-aliasing rules [-Wstrict-aliasing]
```

Aliasing warnings with gcc, cont

```
int len;
struct sockaddr_linx to; //  Definition of struct on next slide
socklen_t socklen;
int sd = linx_get_descriptor(linx);
sig = (void *)buf; //  sig is declared as 'union LINX_SIGNAL *sig', see next slide
socklen = sizeof(struct sockaddr_linx);
to.family = AF_LINX;
to.spid = test_slave;

sig->sigNo = ATTACH_TEST_REQ;
len = sendto(sd, sig, sizeof(LINX_SIGSELECT), 0,   // - row 107
       (const struct sockaddr *)(void *)&to, socklen); // - row 108
```

Aliasing warnings with gcc, cont

```
union LINX_SIGNAL {
  LINX_SIGSELECT sigNo;
};
...
/* The LINX sockaddr structure. */
struct sockaddr_linx {
  sa_family_t family; /* Shall be set to AF_LINX */
  LINX_SPID spid;   /* An illegal spid is set to 0, other values
         * are considered legal. */
};
...
union LINX_SIGNAL *sig,
...
static char buf[65536]; /* buffer used when not using linx api. */
...
```

LINX_SPID is just typedef:ed to uint32_t.

Aliasing warnings with gcc, cont

I beleive the warning is from variable 'sig' or from variable 'to', probably 'sig', since it points to a union and is used in type-punning (from a char-buffer 'buf)'.

Cast involving (struct sockaddr*) are bread-and-butter of socket coding everywhere, so if that is the culprit, the example is maybe not optimal. But it is good to know that gcc is still evolving, 4.7.x does more on aliasing than 4.1.x.

Excerpt from release notes of gcc-4.5, under 'General Optimizer Improvements':

- The infrastructure for optimizing based on restrict qualified pointers has been rewritten and should result in code generation improvements. *Optimizations based on restrict qualified pointers are now also available when using -fno-strict-aliasing*. (my italics)

Advice from the experts

Simplify expressions. Do not mix memory access with calculations. Use the [Load --> Update --> Store] pattern.

Advice, continued

- Strict aliasing means that two objects of different types cannot refer to the same location in memory. Enable this option in GCC with the -fstrict-aliasing flag. Be sure that all code can safely run with this rule enabled. Enable strict aliasing related warnings with -Wstrict-aliasing, *but do not expect to be warned in all cases*.
- Compare the assembly output of the function with restricted pointers and file scope arrays to ensure that all of the possible aliasing information has been used.
- Only use restricted leaf pointers. Use of parent pointers may break the restrict contract.
- Publish as many assumptions as possible about aliasing information in the function declaration.

Advice, continued

- Memory windows may be overlapping and still be without aliases. Do not limit the data design to non-overlapping windows.
- Begin using the restrict keyword immediately. Retrofit old code as soon as possible.
- Keep loads and stores separated from calculations. This results in better scheduling in GCC, and makes the relationship between the output assembly and the original source clearer.

My own thoughts

- Trying to produce aliasing-clean code using unions is possible, but can be frustrating. Aliasing is still not mentioned in most C/C++ books!
- C and C++ are fundamentally byte-oriented languages. E.g. the sizeof () operator returns size in bytes, a char* can alias anything...
- Most modern archictectures are *not* byte-oriented, e.g. use 32-bit or even wider loads and stores. The buses in the mailbox project are endianness-transparent and work only with 32-bit loads and stores.

Aaargh! I still want to do this:

Maybe Linus Torvalds simply is right, as usual? We just want to do our work and not have the compiler clusterf**k our code...

```
int Mailbox_GetMessage(
    union Message** msg,
    uint16_t* msgSize,
    uint8_t* source,
    uint8_t* dest)
{
   uint32_t* buf;
     ...
   *msg = (union Message*)&buf[1];
     // buf[0] is header, invisible to user

   return 0; // Processing OK
}
```

Ideas?

Sometimes, we want to use C or C++ as portable assembler.

Maybe we could have a switch that tells the compiler '*do not optimize this function, just do a straight translation to assembler, whatever that is. And warn me and maybe hint of ways of making the code faster, but do not change anything. If the C/C++ code is stupid, the compiler should just translate it anyway.*'

The compiler could e.g. hint that a 'restrict' on some vars/parameters may help it do a straighter translation. Maybe something for the clang developers?

So, how did all this affect the design of the mailbox API?

- Disclaimer: Still work in progress, we modified the mailbox internals yesterday to remove gcc 4.7 warnings...
- Some design similarities to Enea LINX open-source API. Thank you, LINX hackers. It is always nice to learn from prior art, good code by good programmers!
- No memcpy() on target (used in unit and integration tests on our workstations, though)
- union Message {} used for payload part of messages, API handles Message*, makes strict aliasing much less of an issue for clients
- As already mentioned, no bitfields
- Avoid macros, use C99/C++ inline functions where deemed necessary

Mailbox API design bullets, cont.

- Mailbox tries handling strict aliasing internally
- union Message contains different messages for different clients, usually small (3-10 word payload) messages
- Receive side of Mailbox internally uses a definition of union Message {} containing an uint32_t buffer, 16*64 uint32_t words. Big! (on our little RISC CPU)
- Send side of mailbox is leaner, but not formally aliasing-free. But gcc 4.7 does not complain.
- Message {} being different in different places makes Lint angry... Had to add some Lint suppressions.
- Interesting design Q: Some experts advocate using char buffers instead, since char* is special in C/C++ and can alias anything. But at least, we are aliasing-free in gcc 4.7 with our uint32_t buffers...

Final thoughts, ramble mode <ON>

Low-level C and C++ programmers need to get in the habit of looking at the assembly code the compiler generates, at least every once in a while. But I am an X86 ASM illiterate, and I hope we can soon kick the arcane, huge, disgusting X86 instruction set onto the scrap-heap of history. Heck, already in the early 80s there were beautiful, clean archs like Motorola 68K...

Give me a fast, reasonably priced MIPS-based big-endian Linux machine that can replace my X86 Linux Thinkpad and I'll read gcc or clang assembly output every day! (of course, I can install a cross compiler, but this is often not practical)

That's all, folks!

Thank you for listening!