Why shouldn't I use ioremap on system memory for ARMv6+?

I need to reserve a large buffer of physically contiguous RAM from the kernel and be able to gaurantee that the buffer will always use a specific, hard-coded physical address. This buffer should remain reserved for the kernel's entire lifetime. I have written a chardev driver as an interface for accessing this buffer in userspace. My platform is an embedded system with ARMv7 architecture running a 2.6 Linux kernel.

Chapter 15 of Linux Device Drivers, Third Edition has the following to say on the topic (page 443):

> Reserving the top of RAM is accomplished by passing a `mem=` argument to the kernel at boot time. For example, if you have 256 MB, the argument `mem=255M` keeps the kernel from using the top megabyte. Your module could later use the following code to gain access to such memory: `dmabuf = ioremap (0xFF00000 /* 255M */, 0x100000 /* 1M */);`

I've done that plus a couple of other things:

1. I'm using the `memmap` bootarg in addition to the `mem` one. The kernel boot parameters documentation suggests always using `memmap` whenever you use `mem` to avoid address collisions.
2. I used `request_mem_region` before calling `ioremap` and, of course, I check that it succeeds before moving ahead.

This is what the system looks like after I've done all that:

```
# cat /proc/cmdline
root=/dev/mtdblock2 console=ttyS0,115200 init=/sbin/preinit earlyprintk debug
mem=255M memmap=1M$255M
# cat /proc/iomem
08000000-0fffffff : PCIe Outbound Window, Port 0
  08000000-082fffff : PCI Bus 0001:01
    08000000-081fffff : 0001:01:00.0
    08200000-08207fff : 0001:01:00.0
18000300-18000307 : serial
18000400-18000407 : serial
1800c000-1800cfff : dmu_regs
18012000-18012fff : pcie0
18013000-18013fff : pcie1
18014000-18014fff : pcie2
19000000-19000fff : cru_regs
1e000000-1fffffff : norflash
40000000-47ffffff : PCIe Outbound Window, Port 1
  40000000-403fffff : PCI Bus 0002:01
    40000000-403fffff : 0002:01:00.0
  40400000-409fffff : PCI Bus 0002:01
    40400000-407fffff : 0002:01:00.0
    40800000-40807fff : 0002:01:00.0
80000000-8feffff : System RAM
  80052000-8045dfff : Kernel text
  80478000-80500143 : Kernel data
8ff00000-8fffffff : foo
```

Everything so far looks good, and my driver is working perfectly. I'm able to read and write directly to the specific physical address I've chosen.

However, during bootup, a big scary warning (™) was triggered:

```
BUG: Your driver calls ioremap() on system memory.  This leads
to architecturally unpredictable behaviour on ARMv6+, and ioremap()
will fail in the next kernel release.  Please fix your driver.
------------[ cut here ]------------
WARNING: at arch/arm/mm/ioremap.c:211 __arm_ioremap_pfn_caller+0x8c/0x144()
Modules linked in:
[] (unwind_backtrace+0x0/0xf8) from [] (warn_slowpath_common+0x4c/0x64)
[] (warn_slowpath_common+0x4c/0x64) from [] (warn_slowpath_null+0x1c/0x24)
[] (warn_slowpath_null+0x1c/0x24) from [] (__arm_ioremap_pfn_caller+0x8c/0x144)
[] (__arm_ioremap_pfn_caller+0x8c/0x144) from [] (__arm_ioremap_caller+0x50/0x58)
[] (__arm_ioremap_caller+0x50/0x58) from [] (foo_init+0x204/0x2b0)
[] (foo_init+0x204/0x2b0) from [] (do_one_initcall+0x30/0x19c)
[] (do_one_initcall+0x30/0x19c) from [] (kernel_init+0x154/0x218)
[] (kernel_init+0x154/0x218) from [] (kernel_thread_exit+0x0/0x8)
---[ end trace 1a4cab5dbc05c3e7 ]---
```

Triggered from: arc/arm/mm/ioremap.c

```
/*
 * Don't allow RAM to be mapped - this causes problems with ARMv6+
```

```
 */
if (pfn_valid(pfn)) {
    printk(KERN_WARNING "BUG: Your driver calls ioremap() on system memory.  This
leads\n"
           KERN_WARNING "to architecturally unpredictable behaviour on ARMv6+, and
ioremap()\n"
           KERN_WARNING "will fail in the next kernel release.  Please fix your
driver.\n");
    WARN_ON(1);
}
```

What problems, exactly, could this cause? Can they be mitigated? What are my alternatives?

linux linux-kernel arm embedded-linux ioremap

edited Mar 31 at 16:06

asked Mar 30 at 20:33

Woodrow Barlow

2,205 ● 2 ● 7 ● 38

One alternative on modern, devicetree-based embedded systems is to reserve memory for the device in the devicetree. See "Documentation/devicetree/bindings/reserved-memory/reserved-memory.txt" in the kernel sources. – Ian Abbott Mar 31 at 11:54

1 Answer

> So I've done exactly that, and it's working.

Provide the kernel command line (e.g. **/proc/cmdline**) and the resulting memory map (i.e. **/proc/iomem**) to verify this.

> What problems, exactly, could this cause?

The problem with using **ioremap()** on system memory is that you end up assigning conflicting attributes to the memory which causes "unpredictable" behavior.
See the article "ARM's multiply-mapped memory mess", which provides a history to the warning that you are triggering.

> The ARM kernel maps RAM as normal memory with writeback caching; it's also marked non-shared on uniprocessor systems. The ioremap() system call, used to map I/O memory for CPU use, is different: that memory is mapped as device memory, uncached, and, maybe, shared. These different mappings give the expected behavior for both types of memory. Where things get tricky is when somebody calls ioremap() to create a new mapping for system RAM.
>
> The problem with these multiple mappings is that they will have differing attributes. As of version 6 of the ARM architecture, the specified behavior in that situation is "unpredictable."

Note that "system memory" is the RAM that is managed by the kernel.
The fact that you trigger the warning indicates that your code is generating multiple mappings for a region of memory.

> Can they be mitigated?

You have to ensure that the RAM you want to **ioremap()** is not "system memory", i.e. managed by the kernel.
See also this answer.

ADDENDUM

This warning that concerns you is the result of **pfn_valid(pfn)** returning TRUE rather than FALSE.
Based on the Linux cross-reference link that you provided for version 2.6.37, **pfn_valid()** is simply returning the result of

```
memblock_is_memory(pfn << PAGE_SHIFT);
```

which in turn is simply returning the result of

```
memblock_search(&memblock.memory, addr) != -1;
```

I suggest that the kernel code be hacked so that the conflict is revealed.
Before the call to **ioremap()**, assign TRUE to the global variable `memblock_debug`.

The following patch should display the salient information about the memory conflict.
(The memblock list is ordered by base-address, so **memblock_search()** performs a binary search on this list, hence the use of `mid` as the index.)

```
 static int __init_memblock memblock_search(struct memblock_type *type, phys_addr_t
addr)
 {
         unsigned int left = 0, right = type->cnt;

         do {
                 unsigned int mid = (right + left) / 2;

                 if (addr < type->regions[mid].base)
                         right = mid;
                 else if (addr >= (type->regions[mid].base +
                                   type->regions[mid].size))
                         left = mid + 1;
-                else
+                else {
+                        if (memblock_debug)
+                                pr_info("MATCH for 0x%x: m=0x%x b=0x%x s=0x%x\n",
+                                        addr, mid,
+                                        type->regions[mid].base,
+                                        type->regions[mid].size);
                         return mid;
+                }
         } while (left < right);
         return -1;
 }
```

If you want to see all the memory blocks, then call **memblock_dump_all()** with the variable `memblock_debug` is TRUE.

[Interesting that this is essentially a programming question, yet we haven't seen any of your code.]

ADDENDUM 2

Since you're probably using ATAGs (instead of Device Tree), and you want to dedicate a memory region, fix up the ATAG_MEM to reflect this smaller size of physical memory. Assuming you have made zero changes to your boot code, the ATAG_MEM is still specifying the full RAM, so perhaps this could be the source of the system memory conflict that causes the warning.
See this answer about ATAGs and this related answer.

edited May 23 at 12:09
Community ♦
1 ● 1

answered Mar 31 at 5:39
sawdust
7,371 ● 15 ● 32

i've added the details you requested -- it seems like the RAM i've mapped *isn't* system memory. – Woodrow Barlow Mar 31 at 13:19

or, rather, that i've successfully taken it away from the system. – Woodrow Barlow Mar 31 at 13:58

Added advice for you to try. – sawdust Mar 31 at 19:46

i'll try this out and report back in the morning. i left out my code because driver code can get pretty lengthy and i didn't think it would help (since the technique was already described and this is really about Kernel internals and ARM specs). if you disagree, i can clean it up and post it. – Woodrow Barlow Mar 31 at 19:57

Came up with a patch that should provide more info, and also a possible root cause. – sawdust Mar 31 at 21:59

i'm going to mark this answer, because you've given me everything i need to know that what i'm doing is not going to cause architecturally unstable behavior. i'm still investigating ATAGs and DeviceTree as alternative methods for marking reserved memory. – Woodrow Barlow Apr 3 at 15:14

Device Tree was gradually introduced for ARM in versions 3.x. Since you're using 2.6 there's only ATAGs. – sawdust Apr 4 at 1:44

unfortunately, i'm not using UBOOT. it looks like ATAGs will still be possible, but it will take some work. – Woodrow Barlow Apr 4 at 16:08

Regardless of how it is loaded, the Linux ARM 2.6 kernel *requires* ATAGs. And thanks for the bonus. – sawdust Apr 4 at 19:12

okay, that's useful to know. i just have to find it now. – Woodrow Barlow Apr 4 at 19:15