WIKIPEDIA

Duff's device

In the C programming language, **Duff's device** is a way of manually implementing loop unrolling by interleaving two syntactic constructs of C: the `do-while` loop and a switch statement. Its discovery is credited to Tom Duff in November 1983, when Duff was working for Lucasfilm and used it to speed up a real-time animation program.

Loop unrolling attempts to reduce the overhead of conditional branching needed to check whether a loop is done, by executing a batch of loop bodies per iteration. To handle cases where the number of iterations is not divisible by the unrolled-loop increments, a common technique among assembly language programmers is to jump directly into the middle of the unrolled loop body to handle the remainder.[1] Duff implemented this technique in C by using C's case label fall-through feature to jump into the unrolled body.[2]

Contents

Original version

Duff's problem was to copy 16-bit units ("shorts" in most C implementations) from an array into a memory-mapped output register, denoted in C by a pointer. His original code, in K&R C, looked as follows:[3][4]

```
send(to, from, count)
register short *to, *from;
register count;
{
    do {                          /* count > 0 assumed */
        *to = *from++;
    } while(--count > 0);
}
```

This code assumes that initial `count > 0`. The pointer `to` is not incremented as would be required for a memory-to-memory copy. If `count` were always divisible by eight, unrolling this loop eight-fold would produce the following:

```
send(to, from, count)
register short *to, *from;
register count;
{
    register n = count / 8;
    do {
        *to = *from++;
        *to = *from++;
        *to = *from++;
        *to = *from++;
        *to = *from++;
        *to = *from++;
        *to = *from++;
        *to = *from++;
```

```
    } while (--n > 0);
}
```

Duff realized that to handle cases where `count` is not divisible by eight, the assembly programmer's technique of jumping into the loop body could be implemented by interlacing the structures of a switch statement and a loop, putting the switch's `case` labels at the points of the loop body that correspond to the remainder of `count / 8`:[1]

```
send(to, from, count)
register short *to, *from;
register count;
{
    register n = (count + 7) / 8;
    switch (count % 8) {
    case 0: do { *to = *from++;
    case 7:      *to = *from++;
    case 6:      *to = *from++;
    case 5:      *to = *from++;
    case 4:      *to = *from++;
    case 3:      *to = *from++;
    case 2:      *to = *from++;
    case 1:      *to = *from++;
            } while (--n > 0);
    }
}
```

Duff's device can similarly be applied with any other size for the unrolled loop, not just eight as in the example above.

Mechanism

Based on an algorithm used widely by programmers coding in assembly for minimizing the number of tests and branches during a copy, Duff's device appears out of place when implemented in C. The device is valid C by virtue of two attributes in C:

1. Relaxed specification of the `switch` statement in the language's definition. At the time of the device's invention this was the first edition of *The C Programming Language* which requires only that the body of the `switch` be a syntactically valid (compound) statement within which `case` labels can appear prefixing any sub-statement. In conjunction with the fact that, in the absence of a `break` statement, the flow of control will *fall through* from a statement controlled by one `case` label to that controlled by the next, this means that the code specifies a succession of `count` copies from sequential source addresses to the memory-mapped output port.
2. The ability to jump into the middle of a loop in C.

This leads to what the *Jargon File* calls "the most dramatic use yet seen of fall through in C".[5] C's default fall-through in case statements has long been one of its most controversial features; Duff himself said that "This code forms some sort of argument in that debate, but I'm not sure whether it's for or against."[6]

Simplified explanation

The basic idea of loop unrolling is that the number of instructions executed in a loop can be reduced by reducing the number of loop tests, sometimes reducing the amount of time spent in the loop. For example, in the case of a loop with only a single instruction in the block code, the loop test will typically be performed for every iteration of the loop, that is every time the instruction is executed. If, instead, eight copies of the same instruction are placed in the loop, then the test will be performed only every eight iterations, and this may gain time by avoiding seven tests. However, this only handles a multiple of eight iterations, requiring something else to handle any remainder of iterations.[1]

Duff's device provides a solution by first performing the remainder of iterations, followed by iterating as many times as necessary the multiple of eight similar instructions. To determine the number of remainder iterations, the code first calculates the total number of iterations modulo eight. According to this remainder, the program execution will then *jump* to a `case` statement followed by *exactly the number of iterations needed*. Once this is done, everything is straightforward – the code continues by doing iterations of groups of eight instructions, this has become possible since the remaining number of iterations is a multiple of eight.[1]

Duff's device provides a compact loop unrolling by using the case keyword *both inside and outside the loop*. This is unusual because the contents of a case statement are traditionally thought of as a block of code nested inside the case statement, and a reader would typically expect it to end before the next case statement. According to the specifications of C language, this is not necessary; indeed, case statements can appear anywhere inside the switch code block, and at any depth; the program execution will simply jump to the next statement, wherever it may be.

A functionally equivalent version with `switch` and `while` disentangled

```
send(to, from, count)
register short *to, *from;
register count;
{
    register n = (count + 7) / 8;
    switch (count % 8) {
        case 0: *to = *from++;
        case 7: *to = *from++;
        case 6: *to = *from++;
        case 5: *to = *from++;
        case 4: *to = *from++;
        case 3: *to = *from++;
        case 2: *to = *from++;
        case 1: *to = *from++;
    }
    while (--n > 0) {
        *to = *from++;
        *to = *from++;
        *to = *from++;
        *to = *from++;
        *to = *from++;
        *to = *from++;
        *to = *from++;
        *to = *from++;
    }
}
```

Performance

Many compilers will optimize the switch into a jump table just as would be done in an assembly implementation.

The primary increase in speed versus a simple, straightforward loop, comes from loop unwinding that reduces the number of performed branches, which are computationally expensive due to the need to flush—and hence stall—the instruction pipeline. The `switch` statement is used to handle the remainder of the data not evenly divisible by the number of operations unrolled (in this example, eight byte moves are unrolled, so the `switch` handles an extra 1–7 bytes automatically).

This automatic handling of the remainder may not be the best solution on all systems and compilers – in some cases two loops may actually be faster (one loop, unrolled, to do the main copy, and a second loop to handle the remainder). The problem appears to come down to the ability of the compiler to correctly optimize the device; it may also interfere with pipelining and branch prediction on some architectures.[6] When numerous instances of Duff's device were removed from the XFree86 Server in version 4.0, there was an improvement in performance and a noticeable reduction in size of the executable.[7] Therefore, when considering this code as a program optimization, it may be worth running a few benchmarks to verify that it actually is the fastest code on the target architecture, at the target optimization level, with the target compiler, as well as weighing the risk that the optimized code will later be used on different platforms where it is not the fastest code.

For the purpose of memory-to-memory copies (which, as mentioned above, was not the original use of Duff's device), the standard C library provides function `memcpy`[8]; it will not perform worse than a memory-to-memory copy version of this code, and may contain architecture-specific optimizations that will make it significantly faster.[9][10]

See also

- Coroutine – Duff's device can be used to implement coroutines in C/C++
- Jensen's Device

Notes

References

1. Ralf Holly (August 1, 2005). "A Reusable Duff Device" (http://www.drdobbs.com/a-reusable-duff-device/184406208?queryText=%2522duff%2527s%2Bdevice%2522). *Dr. Dobb's Journal*. Retrieved September 18, 2015.
2. Tom Duff (August 29, 1988). "Subject: Re: Explanation, please!" (https://www.lysator.liu.se/c/duffs-device.html). *Lysator*. Retrieved November 3, 2015.
3. "The amazing Duff's Device by Tom Duff!" (http://doc.cat-v.org/bell_labs/duffs_device). *doc.cat-v.org*. Retrieved 2017-06-08.

4. Cox, Russ (2008-01-30). "research!rsc: On Duff's Device and Coroutines" (https://research.swtch.com/duff). *research.swtch.com*. Retrieved 2017-01-24.
5. The Jargon File (http://www.catb.org/jargon/html/D/Duffs-device.html)
6. James Ralston's USENIX 2003 Journal (http://www.l33tskillz.org/usenix2003/notes/t09-5/)
7. Ted Tso (August 22, 2000). "Re: [PATCH] Re: Move of input drivers, some word needed from you" (http://lkml.indiana.edu/hypermail/linux/kernel/0008.2/0171.html). *lkml.indiana.edu*. Linux kernel mailing list. Retrieved August 22, 2014. "Jim Gettys has a wonderful explanation of this effect in the X server. It turns out that with branch predictions and the relative speed of CPU vs. memory changing over the past decade, loop unrolling is pretty much pointless. In fact, by eliminating all instances of Duff's Device from the XFree86 4.0 server, the server shrunk in size by _half_ _a_ _megabyte_ (!!!), and was faster to boot, because the elimination of all that excess code meant that the X server wasn't thrashing the cache lines as much."
8. "memcpy - cppreference.com" (http://en.cppreference.com/w/c/string/byte/memcpy). En.cppreference.com. Retrieved 2014-03-06.
9. Wall, Mike (2002-03-19). "Using Block Prefetch for Optimized Memory Performance" (http://web.mit.edu/ehliu/Public/ProjectX/Meetings/AMD_block_prefetch_paper.pdf) (PDF). mit.edu. Retrieved 2012-09-22.
10. Fog, Agner (2012-02-29). "Optimizing subroutines in assembly language" (http://www.agner.org/optimize/optimizing_assembly.pdf) (PDF). Copenhagen University College of Engineering. pp. 100 ff. Retrieved 2012-09-22.

Further reading

- Kernighan, Brian; Ritchie, Dennis (March 1988). *The C Programming Language* (2nd ed.). Englewood Cliffs, N.J.: Prentice Hall. ISBN 0-13-110362-8.

External links

- Description and original mail by Duff at Lysator
- Simon Tatham's coroutines in C utilizes the same switch/case trick
- Adam Dunkels' Protothreads - Lightweight, Stackless Threads in C also uses nested switch/case statements (see also The lightest lightweight threads, Protothreads)
- Pigeon's device is a related technique: intertwined switch/case and if/else statements

Retrieved from "https://en.wikipedia.org/w/index.php?title=Duff%27s_device&oldid=884527648"

This page was last edited on 22 February 2019, at 06:13 (UTC).

Text is available under the Creative Commons Attribution-ShareAlike License; additional terms may apply. By using this site, you agree to the Terms of Use and Privacy Policy. Wikipedia® is a registered trademark of the Wikimedia Foundation, Inc., a non-profit organization.