Compiler optimization about elimination of pointer operation on inline function in C?

Asked 10 years ago Active 11 months ago Viewed 2k times

6

3

If this function `Func1` is inlined,

```
inline int Func1 (int* a)
{
    return *a + 1;
}

int main ()
{
    int v = GetIntFromUserInput(); // Unknown at compile-time.
    return Func1(&v);
}
```

Can I expect a smart compiler to eliminate the pointer operations? (`&a` and `*a`) As I guess, the function will be transformed into something like this,

```
int main ()
{
    int v = GetIntFromUserInput(); // Unknown at compile-time.
    int* a = &v;
    return *a + 1;
}
```

and finally,

```
int main ()
{
    int v = GetIntFromUserInput(); // Unknown at compile-time.
    return v + 1;
}
```

Pointer operations look easily being eliminated. But I heard that pointer operation is something special and cannot be optimized.

c optimization pointers compiler-construction

Share Follow

edited Jun 14 '18 at 1:56

asked Mar 12 '11 at 6:46

eonil

74.7k 72 292 479

Did you look at the output from your compiler? Your question seems to be an exact duplicate of stackoverflow.com/questions/2021141/…. – Carl Norum Mar 12 '11 at 6:49

@Carl Norum: While the question itself might be similar the answers here seem very very useful. – the_drow Mar 12 '11 at 8:04

1 Sorry, but to me, the time spent wondering if this would work would be more than the time to write `#define Func1(a)(*`

`(a)+1)` which you can be sure would be optimized. – Mike Dunlavey Mar 12 '11 at 14:52

1 @Mike I did thought about it, however macros are not visible to compiler and debugger, it's too hard to debug. And inline functions can be not inlined easily by compiler options. So I have decided to evade using macros to replace functions which are important semantically. (macros only for textual replacement) However I'm feeling your opinion is pretty attractive and considerable. – eonil Mar 14 '11 at 2:49

For a macro this size, seems to me it's a non-issue. OTOH sometimes I've written some pretty big multi-line macros (which save a lot of code and can't be made into functions). Then, if I need to debug, I instantiate the macro by hand and step through it, fix it, and put the macro back. I know that's dirty, but nobody said software is always clean. – Mike Dunlavey Mar 14 '11 at 21:31

2 Answers

Active | Oldest | Votes

5

It is reasonable that it might occur. For example, `gcc -O3` does so:

```
.globl main
        .type   main, @function
main:
        pushl   %ebp
        movl    %esp, %ebp
        andl    $-16, %esp
        call    GetIntFromUserInput
        movl    %ebp, %esp
        popl    %ebp
        addl    $1, %eax
        ret
```

Notice that it takes the return value from the function, adds one, and returns.

Interestingly, it also compiled a Func1, probably since `inline` seems like it should have the meaning of `static`, but an external function (like GetIntFromUserInput) ought to be able to call it. If I add `static` (and leave `inline`), it does remove the function's code.

Share Follow

answered Mar 12 '11 at 6:55

wallyk
53.7k 14 77 133

Thanks. I checked result with `otool`. Can I know the what your tool is? Print result looks differently a little. – eonil Mar 12 '11 at 7:36

I used `gcc -S -O3 t.c` and examined t.s, the generated assembly language. `-S` requests compilation to assembly language only—it does not assemble or link. This was from `gcc (GCC) 4.4.4 20100630 (Red Hat 4.4.4-10)` targeting x86. – wallyk Mar 12 '11 at 21:19

Note that removing `inline` makes no difference in the code that gcc produces. `inline` is ignored and the compiler produces the best code it can according to constraints imposed by visibility. – wallyk Mar 13 '11 at 19:46

8

Yes the compiler, as said by Wallyk, is able to remove useless operations in this case.

However you must remember that when you specify a function signature something is lost in the translation from your problem domain to C. Consider the following function:

```
void transform(const double *xyz, // Source point
               double *txyz,      // Transformed points
               const double *m,   // 4x3 transformation matrix
               int n)             // Number of points to transform
{
    for (int i=0; i<n; i++) {
        txyz[0] = xyz[0]*m[0] + xyz[1]*m[3] + xyz[2]*m[6] + m[9];
        txyz[1] = xyz[0]*m[1] + xyz[1]*m[4] + xyz[2]*m[7] + m[10];
        txyz[2] = xyz[0]*m[2] + xyz[1]*m[5] + xyz[2]*m[8] + m[11];
        txyz += 3; xyz += 3;
    }
}
```

I think that the intent is clear, however the compiler must be paranoid and consider that the generated code must behave exactly as described by the C semantic even in cases that are of course not part of the original problem of transforming an array of points like:

- `txyz` and `xyz` are pointing to the same memory address, or maybe they are pointing to adjacent doubles in memory
- `m` is pointing inside the `txyz` area

This means that for the above function the C compiler is forced to assume that after each write to `txyz` any of `xyz` or `m` could change and so those values cannot be loaded in free order. The resulting code consequently will not be able to take advantage of parallel execution for example of the computations of the tree coordinates even if the CPU would allow to do so.

This case of aliasing was so common that C99 introduced a specific keyword to be able to tell the compiler that nothing so strange was intended. Putting the `restrict` keyword in the declaration of `txyz` and `m` reassures the compiler that the pointed-to memory is not accessible using other ways and the compiler is then allowed to generate better code.

However this "paranoid" behavior is still necessary for all operations to ensure correctness and so for example if you write code like

```
char *s = malloc(...);
char *t = malloc(...);
... use s and t ...
```

the compiler has no way to know that the two memory areas will be non-overlapping or, to say it better, there is no way to define a signature in the C language to express the concept that returned values from `malloc` are "non overlapping". This means that the paranoid compiler (unless some non-standard declarations are present for malloc and the compiler has a special handling for it) will think in the subsequent code that any write to something pointed by `s` will possibly overwrite data pointed by `t` (even when you're not getting past the size passed to `malloc` I mean ;-)).

In your example case even a paranoid compiler is allowed to assume that

1. no one will know the address of a local variable unless getting it as a parameter
2. no unknown external code is executed between the reading and computation of addition

If both those points are lost then the compiler must think to strange possibilities; for example

```
int a = malloc(sizeof(int));
*a = 1;
printf("Hello, world.\n");
// Here *a could have been changed
```

This crazy thought is necessary because `malloc` knows the address of `a`; so it could have passed this information to `printf`, which after printing the string could use that address to change the content of the location. This seems clearly absurd and maybe the library function declaration could contain some special unportable trick, but it's necessary for correctness in general (imagine `malloc` and `printf` being two user defined functions instead of library ones).

What does all this blurb mean? That yes, in your case the compiler is allowed to optimize, but it's very easy to remove this possibility; for example

```
inline int Func1 (int* a) {
    printf("pointed value is %i\n", *a);
    return *a + 1;
}

int main () {
    int v = GetIntFromUserInput();   // Assume input value is non-determinable.
    printf("Address of v is %p\n", &v);
    return Func1(&v);
}
```

is a simple variation of your code, but in this case the compiler cannot avoid assuming that the second `printf` call could have changed the pointed memory even if it's passed just the pointed value and not the address (because the first call to `printf` was passed the address and so the compiler must assume that potentially that function could have stored the address to use it later to alter the variable).

A very common misconception in C and C++ is that liberal use of the keyword `const` with pointers or (in C++) references will help the optimizer generating better code. This is completely false:

1. In the declaration `const char *s` the nothing is said about that the pointed character is going to be constant; it's simply said that it is an error to change the pointed character **using that pointer**. In other words `const` in this case simply means that the pointer is "readonly" but doesn't tell that, for example, other pointers could be used to changed the very same memory pointed to by `s`.
2. It is legal in C (and C++) to "cast away" const-ness from a pointer (or reference) to constant. So the paranoid compiler must assume that even a function has been only handed a `const int *` the function could store that pointer and later can use it to change the memory pointed to.

The `const` keyword with pointers (and C++ references) is only meant as an aid for the programmer to avoid unintentional writing use of a pointer that was thought as being used only for reading. Once this check is performed then this `const` keyword is simply forgotten by the optimizer because it has no implications in the semantic of the language.

Sometimes you may find another silly use of the `const` keyword with parameters that tells that the value of the parameter cannot be changed; for example `void foo(const int x)`. This kind of use has no real philosophical meaning for the signature and simply puts some little annoyance on the implementation of the called function: a parameter is a copy of a value and caller shouldn't care if the called function is going to

change that copy or not... the called function can still make a copy of the parameter and change that copy so nothing is gained anyway.

To recap... when the compiler sees

```
void foo(const int * const x);
```

must still assume that foo will potentially store away a copy of the passed pointer and that can use this copy to change the memory pointed to by `x` immediately or later when you call any other unknown function.

This level of paranoia is required because of how the language semantic is defined.

It is very important to understand this "aliasing" problem (there can be different ways to alter the same writable area of memory), especially with C++ where there is a common anti-pattern of passing around const references instead of values even when logically the function should accept a value. See this answer if you are also using C++.

All these are the reasons for which when dealing with pointers or references the optimizer has much less freedom than with local copies.

Share Follow

edited Apr 23 '20 at 8:08

answered Mar 12 '11 at 7:57

6502
104k 14 141 250

Wow! Thanks for details. It's far more complicated than I thought... How do you think about if the function `printf` was accept single argument `int const * const a` ? – eonil Mar 13 '11 at 2:30

1 @Eonil: It makes no difference. I extended the answer with a section discussing `const` -ness of pointers. – 6502 Mar 13 '11 at 7:48

Thanks for answer. I stopped depending on optimization assumption :) – eonil Mar 13 '11 at 8:07

The gcc malloc attribute allow the compiler to assume that malloc's returned memory is not aliased by printf, another malloc, or anything else. – Edward Brey Mar 6 '12 at 17:40

@EdwardBrey: That's why I wrote <<"the compiler has no way to know that the two memory areas will be non-overlapping or, to say it better, there is no way to define a signature **in the C language** to express the concept that returned values from malloc are "non overlapping">> and why I also added that compilers can use unportable tricks to do that <<...and may be the library function declaration could contain some special unportable trick>>. Things like that or the type checking done at compile time on `printf` format strings are very useful, but non-standard. – 6502 Mar 6 '12 at 18:46