Make your programs run faster: avoid function calls

June 12, 2020Help the Compiler, Low Level Performance, Performance, Toolchain and Performance2 Replies

*We at **Johnny's Software Lab LLC** are experts in performance. If performance is in any way concern in your software project, feel free to contact us.*

This is the second article in the series related to low-level optimizations, the first being Make your programs run faster by better using the data cache.

Modern software design likes layers, abstractions and interfaces. These concepts have been introduced into programming with best intentions in mind since they allow developers to write software which is easier to understand and maintain. In the world of compilers all these constructs translate to calls to functions: many small functions that call one another while the data gradually moves from one layer to another.

The problem with this concept is that function calls in principle are not cheap operations. To make a call, the program needs to put the call arguments on the program stack or into registers. It also needs to save some of its own registers because they can get overwritten by the called function. The called function isn't necessarily in the instruction cache which can lead to delays in execution and lower performance. When the called function finishes execution, there is also a performance penalty for returning back to the original function.

On one hand, functions are great as a concept that makes software more readable a easier to maintain. On the other hand, too much calls to tiny functions doing little work can definitely slow things down.

Table Of Contents

Techniques to avoid function calls

Let's go through some of the techniques to avoid function calls.

Inlining

Inlining is a techniques used by the compilers to avoid function calls and save some time. Simply put, to inline a function means to put the body of the called function in the place where the call is made. An example:

```
void sort(int* a, int n) {
    for (int i = 0; i < n; i++) {
        for (int j = i; j < n; j++) {
            swap_if_less(&a[i], &a[j]);
        }
    }
}
template <typename T>
void swap_if_less(T* x, T* y) {
    if (*x < *y) {
        std::swap(*x, *y);
    }
}
```

Function `sort` is doing the sorting, and function `swap_if_less` is a helper function used by `sort`. Function `swap_if_less` is a small function and is called from sort many times, so the best way to avoid this is to copy the body of `swap_if_less` into the function `sort` and avoid all overhead related to function calls.

Inlining is normally done by the compiler but you can do it manually as well. We already covered manual inlining, now let's cover inlining made by the compiler. All compilers will by default inline calls to small functions, but there are obstacles to this:

- If a called function is defined in another .C or .CPP file, it cannot be automatically inlined, unless link time optimizations are enabled.
- In C++, if class method is defined in the class declaration, it will be inlined unless too large.
- Functions marked as `static` will probably be automatically inlined.
- If a C++ method is marked as virtual, it cannot be automatically inlined (there are exceptions to this).
- If a function is called using a function pointer, it cannot be inlined. On the other hand, if a function is called as a lambda expression, it can probably be inlined.
- If a function is to long, the compiler will probably not inline it. This decision is motivated by performance, long function is not worth inlining because the function itself takes long time and call overhead is small.

Inlining increases code size, and careless inlining can bring explosion in code size and actually lower performance. So it is best idea to let the compiler decide when and what to inline.

In C and C++ there is a keyword `inline`. If the function declaration has this prefix, that is a recommendation to the compiler to perform the inlining. In practice, compiler uses heuristics to decide which functions to inline and often disregards the hint.

To check if your function is inlined, you can always take a look at the disassembly of the object code made by the compiler (use command `objdump -Dtx my_object_file.o`) or programmatically. GCC and CLANG compilers additionally offer:

- `__attribute__((always_inline))` – forces the compiler to always inline a function. In case it is not possible to inline it, it will generate a compilation warning.
- `__attribute__((flatten))` – if this keyword is in the declaration of a function, all calls made to other functions from that function will be replaced with inline versions (if possible).

Inlining and virtual functions

As we already said, virtual functions cannot be inlined. They are also more expensive than non-virtual functions. A few solutions exist to mitigate this problem:

- If a virtual function simply returns a value, consider keeping this value as a member variable of the base class and initializing it on class construction. Later, a non-virtual function of the base class can return this value.
- You are probably keeping your objects in a container. Instead of keeping several types of objects in the same container, consider having separate containers per object type. So if you have `base_class` and `child_class1`, `child_class2` and `child_class3`, instead of `std::vector<base_class*>`, you will have three containers `std::vector<child_class1>`, `std::vector<child_class2>` and `std::vector<child_class3>`. This is more involved, but actually a lot faster.

Both of the above approaches will make the function calls inlinable.

Inlining in practice

There are certain circumstances where the inlining pays off and there are circumstances where it doesn't. Number one indicator if inlining should be performed is the function size: the smaller the function, the more sense does it make to inline it. If calling a function takes 50 cycles and the function body takes 20 cycles to execute, inlining is perfectly justified. On the other hand, if a function takes 5000 cycles to execute, for each call you will save 1% of the running time which is probably not worth it.

The second criterion for inlining is how many times the function is called. If it is called a few times, it's not much of a save to inline it. On the other hand, if it is called many-many times, it is reasonable to inline it. Yet, remember, even if it is called many times, the performance gains you get by inlining will probably not be worth the effort.

Compilers and linkers know exactly the size of your function and they can make an excellent decision whether to inline it. As far as call frequency is concerned, compilers and linkers have a limited knowledge about this as well, but it is necessary to profile the program on real world examples in order to get the information about function call frequency. But, as I said, large functions are most likely not good candidates for inlining even if they are called many times.

So in practice, you want to let compiler and linker do the inlining almost all of the time. The only time you will like to explicitly control the inlining process is for performance critical functions.

If by profiling your program you've discovered a function that is performance critical, first you should mark it with a `__attribute__((flatten))` so that the compiler inlines all calls to other functions from that function and its whole code becomes one big function. But even if you did that, that doesn't guarantee that the compiler will actually inline everything. You must sure there are no hindrances to inlining, as already discusses:

- Turn on link-time optimizations to allow code from other modules to be inlined.
- Don't call using a function pointer. In this case you lose some flexibility.
- Don't call using a C++ virtual method. You lose some flexibility, but there are a few ways to work around this problem already mentioned.

The only time you would like to inline a function manually is when the compiler is not able to it automatically. If the automatic inlining fails, the compiler will issue a warning and from that point on you should analyse what is preventing the inlining and fix it or optionally manually inline a function.

Last word on inlining: there are functions you do not want to inline. For your performance-critical function, there are code paths that get executed often. But also, other paths, such as error-processing, get executed rarely. You want to keep those in separate functions in order to decrease the pressure on the instruction cache. Mark those functions with `__attribute__((cold))` to let the compiler know they execute rarely so the compiler can move them away from the hot path.

Avoid recursive functions

Recursive functions are functions that call themselves. And while solutions with recursive functions are normally more elegant, from the aspect of program performance the non-recursive solutions are more efficient. So if you are in need of optimizing a code with recursive functions, there are several things you could do:

- Make sure your recursive function is tail recursive. This will allow compiler to tail-call optimizations on your function and convert call to a function into a jump.
- Convert your recursive function to non-recursive using stack data structure. This will save you some time related to function calling, but it ain't simple business.
- Do more stuff in a single iteration of the function. Example:

```
int factorial(int n) {
    if (n <= 1) {
        return 1;
    } else {
        return n * (n - 1) * factorial(n - 2);
    }
}
```

Common implementation of factorial function decreases the argument by 1 in a subsequent call , in this implementation we do more work in the `factorial()` function but decrease the argument by two in the subsequent call.

Use function attributes to give optimization hints to the compiler

GCC and CLANG offer certain function attributes, which when enabled can help the compiler generate better code. These are two of these that are relevant to the compiler: const attribute and pure attribute.

Attribute `pure` means that results of the function only depend on its input parameters and state of memory. The function doesn't write to memory nor calls any other function that potentially does that.

```
int __attribute__((pure)) sum_array(int* array, int n) {
     int res = 0;
    for (int i = 0; i < n; i++) {
        res += a[i];
    }
    return res;
}
```

The good thing about pure functions is that the compiler can omit calls to the same function with same parameters, or remove calls if the parameters are unused.

Attribute `const` means that the result of the function only depends on its input parameters. Example:

```
int __attribute__((const)) factorial(int n) {
    if (n == 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
```

Every const function is also pure, so everything said about pure function is also valid for const function. Additionally, for const function the compiler can calculate their values during compilation and replace them with a constant instead of actually calling the function.

C++ has keyword `const` for member functions but this means different thing: if a method is marked as const, that means that the method doesn't change the state of the object, but it can modify other memory (this includes printing to a screen). The compiler uses this information to do some optimizations; if the member is const, it the state of the object doesn't need to be reloaded if it is already loaded. Example:

```
class test_class {
    int my_value;
private:
    test_class(int val) : my_value(val) {}
    int get_value() const { return my_value; }
};
```

In the example, method `get_value` doesn't change the state of the class and can be declared as const.

Marking functions as const and pure is especially important if your functions are going to be delivered as a library to other developers. You don't know who these people are and the quality of their code and this will make sure the compiler can optimize away some of the code in case of sloppy programming. Note that many function in standard library have these attributes.

Experiments

ffmpeg – inline vs no-inline

We compiled two version of ffmpeg, the default version with full optimizations, and the second crippled version where we disabled inlining through -fno-inline and -fno-inline-small-functions compiler switches. We compiled ffmpeg with following options:

```
./configure --disable-inline-asm --disable-x86asm --extra-cxxflags='-fno-inline -fno-inline-small-functions' --extra-cflags='-fno-inline -fno-inline-small-functions'
```

So what happened? It seems that inlining is not the source of major performance increase in ffmpeg. Here are the results:

Parameter	Inlining disabled	Inlining enabled
Runtime (s)	291.8s	285s

Performance comparison: inlining enabled vs inlining disabled

Regular compilation (with inlining) was only 2.4% faster than the version with inlining disabled. Disappointing, but let's comment about it. As already mentioned, in order to have real benefits from inlining, you need to have short inlinable functions. If you don't, inlining might not bring any benefit.

We profiled ffmpeg for this specific case, and ffmpeg itself uses `av_flatten` and `av_inline` macros that correspond to flatten and inline attributes in GCC. When these attributes are explicitly set, the -finline and fno-inline switches don't have any effect. I think this is the reason why we see such a small difference in performance.

We also tried applying `flatten` attribute to some of the functions in order to make the conversion faster, but there were no functions where this would bring significant increase in performance since there were no really small functions where this would make sense.

Homemade Testing

So we've tried with ffmpeg and the results were not promising. So in order to better understand the effects of inlining, we created some homemade tests. They are available in [our github repository](). To run them, fetch the repository, go to directory `2020-06-functioncalls` and execute `make sorting_runtimes`.

We took a regular selection sort algorithm and played a bit with inlining to see how inlining effects the performance of the sorting. Here is the selection sort algorithm:

```
void sort_regular(int* a, int len) {
    for (int i = 0; i < len; i++) {
        int min = a[i];
        int min_index = i;
        for (int j = i+1; j < len; j++) {
            if (a[j] < min) {
                min = a[j];
                min_index = j;
            }
        }
        std::swap(a[i], a[min_index]);
    }
}
```

Please note that the algorithm consists of two nested loops. Inside the loops there is an `if` statement checking if element `a[j]` is smaller than element `min`, and if so it stores the value of the new minimum element.

We created four new function based on this implementation. Two of them are calling an inlined version of the function, the other two are calling not-inlined version of the function. I made sure this happens using GCC `__attribute__((always_inline))` and `__attribute__((noinline))`. Two of them are called `sort_[no]inline_small` and they move the inner `if (a[j] < min) { ... }` statement into a separate function. The other two are called `sort_[no]inline_large` and they move the inner loop `for (int j = i + 1; j < len; j++) { ... }` into a separate functions. Here is how those algorithms look like:

```
void sort_[no]inline_small(int* a, int len) {
    for (int i = 0; i < len; i++) {
        int min = a[i];
        int min_index = i;
        for (int j = i+1; j < len; j++) {
            update_min_index_[no]inline(&a[j], j, &min, &min_index);
        }
        std::swap(a[i], a[min_index]);
    }
}
void sort_[no]inline_large(int* a, int len) {
    for (int i = 0; i < len; i++) {
        int smallest = find_min_element_[no]inline(a, i, len);
        std::swap(a[i], a[smallest]);
    }
}
```

We executed all five above functions on the input array of size 40.000 elements. Here are the results:

	Regular	Small Inline	Small Noinline	Large Inline	Large Noinline
Runtime	1829ms	1850ms	3667ms	1846ms	2294ms

Runtime comparison for various type of inlined functions

As you can see, difference between regular, small inline and large inline are all within a margin of measurements. In case of small noinline function, the inner loop was called 400M times and that is visible in the performance. This implementation is two times slower as regular implementation. In case of large inline function we also see performance drop, but this time the decrease is smaller, about 20%. In this case the inner loop was called 40K times which is much smaller than 400M times we had in first example.

Final Words

Function calls are expensive operations as we can see in our measurements, but luckily modern compilers take care of this very well most of the time. The only thing that the developer needs to make sure is that there are no obstacles to inlining, such as disabled link time optimizations or calls to virtual functions. In case of need to optimize performance sensitive code, developer has the ability to manually enforce inlining through compiler attributes.

Other methods mentioned in this article have limited usability, since a function must have special form in order for compiler to be able to apply them. Nevertheless, they shouldn't be completely neglected.

Bonus: How to check if function is inlined in runtime?

If you want to check if the function is inlined, the first that comes to mind is to look at the assembler code produced by the compiler. But you can also do it programmatically during your program execution.

Say you want to check if a specific call is inlined. You would go about like this. Compiler inlines functions, but for those functions that are exported (and almost all function are exported) it needs to maintain a non-inlined addressable function code that can be called from the outside world.

To check if your function `my_function` is inlined, you need to compare the `my_function` function pointer (which is not inlined) to the current value of the PC. Here is how I did it in my environment (GCC 7, x86_64):

```
void * __attribute__((noinline)) get_pc () { return _builtin_return_address(0); }

void my_function() {
    void* pc = get_pc();
    asm volatile("": : :"memory");
    printf("Function pointer = %p, current pc = %p\n", &my_function, pc);
}
void main() {
    my_function();
}
```

If a function is not inlined, difference between the current value of the PC and value of the function pointer should small, otherwise it will be larger. On my system, when `my_function` is not inlined I get the following output:

```
Function pointer = 0x55fc17902500, pc = 0x55fc1790257b
```

If the function is inlined, I get:

```
Function pointer = 0x55ddcffc6560, pc = 0x55ddcffc4c6a
```

For the non-inlined version difference is 0x7b and for the inlined version difference is 0x181f.

Further Read

[Smarter C/C++ inlining with __attribute__((flatten))]

[Agner.org: Software Optimization Resources]

[Implications of pure and constant functions]

Featured image courtesy of https://www.quora.com/What-is-inline-function-with-syntax-and-example