On Function Call Inlining

Author : Andras Tantos

I've heard so many people claim that the main benefit of inlining is that the compiler can save the call and return instructions. Some examples:

http://stackoverflow.com/questions/145838/benefits-of-inline-functions-in-c
http://www.exforsys.com/tutorials/c-plus-plus/inline-functions.html

That's all well and nice, but that's not even scratching the surface of the benefit of inlining.

The main reason inlining is a powerful optimization tool is that, once the function is inlined, the compiler can optimize it with the call-site. Constants can be propagated into the function body. Loops can be more efficiently re-arranged, code can be hoisted, registers can be (more) optimally allocated, essentially all the heavy artillery that an optimizing compiler has can be deployed on the merged function. Furthermore, since the compiler has full information of what's going on inside the function that was called, it doesn't have to make conservative assumptions about side-effects, subsequent function calls, etc.

The effects of all the above are way more impactful than the saved call and return – even if you take the potential branch-misprediction penalty and parameter save operations into consideration.

Some get it right. Some make a note on this and move on.

The down-side of course is that it could increase code-size. Over-use of inlining might generate more (instruction) cache-misses, which can hurt your performance quite a bit. My personal guidelines:

- Small functions should be inlined
- Large functions with only a few call-sites could be inlined
- Large functions, called from a number of places should not be inlined – though a few specific call-sites could be inlined

Of course most of these decisions are made by your compiler already, but checking the results, and on occasion forcing the compiler to bend your way can be beneficial: unless you use profile-guided optimization, the compiler has to make inlining decisions based on a static view of the code, so it can't take for example execution frequency into consideration.

Finally, the biggest enemies of inlining (in C++) are virtual functions: it's very hard for the compiler to see through a virtual function call and realize that you always (or most of the time) call the same virtual function. Providing non-virtual variants and manually calling them in cases when inlining is expected is probably the best way around this problem.