

## Re: Reducing build-times for large projects.

*Source:* [http://coding.derkeiler.com/Archive/C\\_CPP/comp.lang.cpp/2005-03/1286.html](http://coding.derkeiler.com/Archive/C_CPP/comp.lang.cpp/2005-03/1286.html)

---

*From:* Phil Staite ([phil\\_at\\_nospam.com](mailto:phil_at_nospam.com))

*Date:* 03/09/05

Date: Wed, 09 Mar 2005 02:04:27 -0700

DHOLLINGSWORTH2 wrote:

> *I'm sorry, but Inline functions are loaded right along with the rest of the  
> code. If the line before it executed, then the very next inline code has  
> already been loaded.*

Maybe, if your architecture does speculative execution and fetch. It would also need a deep enough pipeline to see that it needs to do that fetch far enough in advance to get it done in time. It would also depend on the nature of the instructions in front of it, and how long it took the cpu to chew through them. Otherwise you're looking at a stall.

> *You are correct that not loading code is faster than loading and waiting for  
> it to execute. The argument is like saying dont push the BP becuase the  
> actual code to Push the BP has to be loaded, SO DOES THE F"N CODE TO CALL  
> THE FUNCTION THAT IS ALREADY IN CACHE!*

We're talking about the code for the function, not the code that leads up to the function. You either have multiple calls to one set of instructions, or just that set of instructions plonked down in the instruction stream in multiple places. (ie you've replaced your calls with the code)

In the non-inlined case you may get lucky and only fetch the call instruction from main memory, then hit in the cache for the actual function's instructions.

In the case of inlined code, you may get luck and hit the cache for the inlined code too – just as in the non-inlined case.

However, I would contend that it is simple probability that if there is only one copy of the code used by all the places that reference it, that is far more likely to be in cache than one particular instance out of many inlined copies. Also that one copy of the function's code is less likely to push other things out of the cache (that you may later want back) than N copies of the same code, from different addresses.

> *LOOP unrolling has been going on for years. and its a lot faster loading  
> that same chunk of code over and over than staying in the same spot doing*

> *all of the compares.*

Loop unrolling "works" because you get to do N iterations worth of the loop code without hitting a conditional/branch. This reduces the per iteration "overhead" of the loop. It also helps in that speculative execution past a conditional/branch can be problematic. So the more work you can get done between those points the better. At some point though you hit diminishing returns on loop unrolling. You still have to load N times as much code the first time through the loop. Generally it all fits in cache so subsequent iterations are free in that respect. However, you can have too much of a good thing. At some point you're pushing enough instructions out of cache to make room for your unrolled loop that it comes back and bite you. IIRC most compilers limit loop unrolling to the low single digits as far as the number of copies they make.

But this whole discussion is just barely anchored in reality. An awful lot depends on real world parameters that vary widely and wildly from system to system, program to program. I was just trying to point out to the OP that when testing reveals something that makes you say "Oh {expletive}! How'd that happen?" you've got to be ready to challenge your assumptions. One common assumption is that inlining improves execution speed. It may, but then again, it may not.