http://blog.atollic.com/optimizing-code-size-with-the-gnu-gcc-compiler-for-stm32-and-other-arm-cortex-m-targets

Optimizing code-size with the GNU gcc compiler for STM32 and other ARM Cortex-M targets

Posted by Magnus Unemyr on Apr 13, 2015 12:05:32 PM

The discussion of code size has been a never ending story in the embedded compiler industry for decades. During many years, the code size of compilers was the only real differentiator, and code size measurements were publicly used in magazine advertisements as well. Remember the times of the 8051? Embedded compiler vendors have even modified their tools to detect when official benchmark test suites were compiled and generated hand optimized machine code in those cases, to produce “incredible” numbers for marketing use in “competitor comparisons”. Go figure.

Luckily, most embedded compilers in use today are of very high quality, and produce code size of very small size difference in real-life projects (the GNU compiler is a good example of this). Still some compiler vendors use misleading figures in their marketing, but that is of little practical consequence in real life for most customers.

One interesting case not too long ago, was when a commercial compiler vendor claimed to have 30-40% smaller code size than the GNU compiler for the same target architecture. As the difference is typically within only a few % in our own testing on real-life projects (where the GNU compiler sometimes win) this caught my attention.

After some testing, it could be concluded that the compiler vendor was correct. Well, sort of. Or actually, not at all. When you compare compiler optimization and code size, you typically compare both compilers at their highest optimization levels. Only then do both compilers try to do their best. At lower optimization levels, there is no standard that defines “how much” a compiler shall optimize on say, optimization level 1, 2 or 3.

No compiler has something called “no optimization”. Even at the lowest optimization level, all compilers do some optimizations. And so, a compiler vendor can choose to be more aggressive and include more optimization algorithms on also the lower optimization levels, or they can be more conservative and chose to optimize less on the lower optimization levels.

In the particular case mentioned above, the compiler vendor had found out that the GNU compiler generated very compact code on the highest optimization levels, but the GNU compiler developers had been very conservative and selected to use less optimizations on the lowest optimization level. And so, by comparing the compilers on the lowest optimization level, instead of using the highest optimization level that is the norm, they could present incredible (and in my mind, rather meaningless) numbers.

And so, history repeats itself, even-though all compilers today are of very high quality with little differences of practical importance to most customers. Memory is cheap today, after all.

A point worth knowing is that the GNU gcc compiler typically generates a lot of machine code in the default debug mode. The optimization level (–Og) is introduced to enable optimized debuggable code, giving up to 30–40% smaller code compared to –O0 while still keeping the code debug friendly. The optimization level –Og is not active by default and must be enabled manually.

Another source of misunderstandings are the code size contribution made by runtime libraries. While the GNU gcc compiler produces highly optimized and compact code, it is true that the accompanying runtime libraries are not as compact as some commercial alternatives. That may result in relatively large code sizes, if care is not taken with the selection of runtime library.

However, this is often not a practical problem in most cases, as commercial GNU gcc based ARM Cortex tools, such as Atollic TrueSTUDIO, are shipped with commercial runtime library alternatives that vastly reduce the code size. A particular "thief in crime" is printf() and its related cousins (such as sprintf), as they contain many formatting options, including floating point formatting, in order to conform to the ANSI-C standard. By using printf() versions better adapted for embedded use, and typically without floating point formatting, the code size contribution made by the runtime library can be massively reduced compared to the standard runtime library that comes with gcc (newlib).

Furthermore, the GNU gcc compiler and ld linker do not perform full dead code and dead data removal by default. This is done automatically by most commercial compilers, and so if full dead code and dead data removal is not explicitly enabled, the linker will not remove all C/C++ functions that are not called for example, quite obviously producing very unfavorable code size comparisons.

A new trick offered by the GNU gcc compiler and the ld linker is LTO (Link Time Optimization). This is also called whole program analysis/optimization by some people in the industry, although that term is already used for something else in gcc.

Traditionally, a compiler optimizes the machine code in the C file it compiles, and it then generates an object file as input to the linker. The linker then takes a number of object files and resolves any cross dependencies and generate a combined binary image with the machine code generated by the compiler for each C/C++ file. In other words, compilers typically optimize the code inside a C/C++ file, but not between different C/C++ files. After all, how could it, since the machine code generation and optimization is in the compiler, and not in the linker, and the compiler compiles a source code file at a time?

Enters LTO. The GNU gcc compiler can now generate an intermediate byte code as output, instead of machine code, and the GNU ld linker can read this from all the object files it is linking. When linking, the toolchain will then combine the intermediate byte code from the different C/C++ files, and finally optimize it at the very end, before it generates optimized machine code.

In other words, the machine code generation is moved from the compilation of a single file to the linking process, making it possible to do whole program optimizations and code resequencing. Therefore, the code is not only optimized inside C/C++ files, but also between them, enabling resequencing and other optimizations that produce more compact code. In my testing, LTO (link time optimization, or whole program analysis) can give up to approximately 8-10% smaller code size, compared to using exactly the same compiler and linker options but without LTO enabled.

And so, to summarize – to produce compact code size with the GNU gcc compiler for ARM Cortex targets, these steps may be of interest:

- Compile with compiler options for smallest code size (-Os) in release mode
- Compile with compiler options for optimized debugging (-Og) in debug mode
- Make use of compact embedded runtime library alternatives (in particular for printf-styled functions), for example offered by Atollic TrueSTUDIO
- Enable full dead code and dead data removal at the linker stage (disabled by default in the standard GNU tools)
- Possibly, experiment with LTO (link time optimization, also known as whole program analysis outside the gcc community) to introduce also cross-file optimization.

All-in-all, the GNU gcc compiler is a very high quality compiler for ARM Cortex-M targets, producing highly optimized and very compact code. However, with the wrong command line options and runtime library selections, matters can quickly deteriorate and poor code size numbers show up. On the other hand, with the right settings, the GNU gcc compiler is an excellent choice for embedded developers using popular Cortex-M devices like STM32, Kinetis, LPC, etc.

Read more on using the GNU gcc/gdb tools (with the Eclipse IDE) for professional ARM Cortex-M development in these white papers:

Topics: ARM Cortex, GNU tools (GCC/GDB), Atollic TrueSTUDIO