TASKING

OPTIMIZING AT APPLICATION SCOPE

INTRODUCTION

Every programmer at some point during development would need to make changes to his or code to make it more efficient, either to deliver faster speeds or to run on less resources. This is what we call code optimization. While there are several ways to tackle code optimization, such as compressing, rearranging, and inlining, we'll be focusing on the compiler built-in transformation techniques that can reduce your memory usage footprint.

While there are several built-in compiler optimization tools readily available to programmers, their potential remains largely untapped. Either due to unfamiliarity or unawareness of these tools, engineers can be reluctant risking deadlines to try untested techniques, especially if they're involved in developing safety-critical systems.

In this white paper we'll discuss a few easy to implement compiler optimization techniques found in the TASKING toolsets. These techniques are available on different scope levels up to the application scope, including MIL-linking, MIL-splitting, inlining, and code compaction, also known as reverse-inlining. While these code optimization techniques can be applied to any software development application, the context into which we'll be presenting is for safety-critical automotive systems.

THE OPTIMIZATION DILEMMA

Most programmers have been there, with the coding process in its final stages after several thousand lines of code have already been written, but then there's the roadblock. The code you spent hours working on can't be used in product because you've ran out of memory. Your first course of action is to start working on optimizations to reduce your code footprint. The problem is, this journey can be filled with many unexpected obstacles, often requiring many sub-phases that suck time out of your development schedule. While you could leverage existing compiler optimization tools, you've got safety critical systems to work with, and safety standard requirements to meet, such as ISO 26262. Can these basic compiler optimization tools meet those industry standard requirements?

The good news is that modern compilers can apply optimizations at all scope levels ranging from a single machine instruction to expression, statement, basic block, function, module, and application scope. The wider the scope range the more you can get out of the compiler's built-in optimization. These optimization techniques tackle memory usage reduction by removing semantically equivalent code fragments, also known as duplicate code removal or functional clone removal. These duplicates in the code can exist due several reasons, including:

- **Copy and Paste Programming**, which is a common practice of code reusability. While this practice yields minimum efforts in terms of code modification it can also produce inefficient code.
- **Code generated by model-based design tools**, which often make use of templates to generate the code, and as a result can lead to similar blocks of code existing throughout different application module's source code.
- **Fined-grained duplicates**, due to different functions often containing small code blocks with identical semantics, where the quality of the source code can deteriorate if these functional clones are replaced by function calls.
- **Compiler generated instruction sequences**, which can result in similar instruction sets being generated throughout the output file. An example would be the function's prologue and epilogue.

In these examples, notice that the first three refer to duplicates in the source code, whereas the last one refers to the duplicates in the compiler generated machine code. The reduction in code size we can obtain from these optimization techniques will depend on several factors, such as the amount of functional clones in your source code, and the characteristics of the microcontroller instruction set architecture on which the code is executed.

COMPILER OPTIMIZATION TECHNIQUES

In most compilers using a traditional build flow, each C module is passed to the compiler. The compiler then generates an assembly module which is subsequently translated by the assembler into a relocatable object file. Then, the linker connects and locates all object files libraries and outputs an object or hex file.

The TASKING compiler includes a powerful toolset that supports additional build flows where multiple C modules are processed simultaneously. The sections below will explain these TASKING build flows in greater detail, including MIL-Linking, MIL-Splitting, Inlining, Code Compaction, and Linker Optimizations.

MIL-Linking

When the MIL-linking feature is enabled, multiple C modules are passed to the C compiler in a single execution. The compiler frontend (FE) will translate each module into the compiler internal representation called MIL, which is an abbreviation for Medium Level Intermediate Format.

Next, the frontend will link the MIL representation of all source modules and then optimize the merged MIL's and perform inter-module optimizations. The optimized MIL representation is then passed to the compiler backend (BE) which translates the MIL into assembly instructions and performs additional target-specific optimizations. Following that, the compiler generates the assembly file, which will contain all the C module code and data. When linked with the run-time library, the floating-point library and C library will still be required as shown in the figure below.

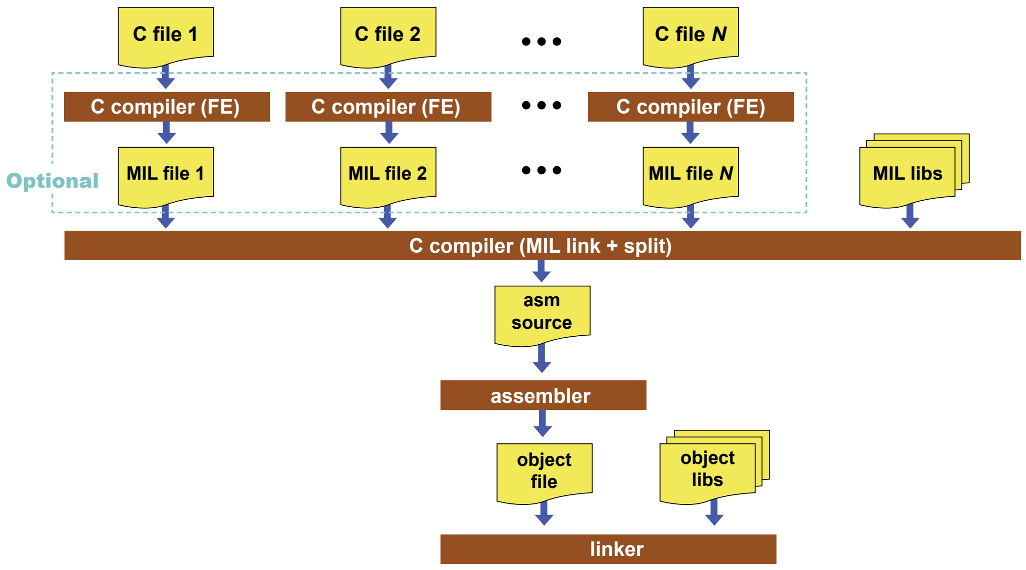

Figure 1: MIL-Link Build Flow

The advantage of this process is that when all files of an application are passed to the compiler then all optimizations can be applied at the application wide scope. This maximizes the opportunity to detect and remove duplicates from the source code and the compiler generated instruction sequences.

There is a drawback to compiling all modules of an application simultaneously. If one C module is modified, then all C modules have to be recompiled. This can significantly slow down the edit-compile-debug cycle. However, this can be mitigated by dividing the modules into multiple sets and applying the "application wide optimization" on a scope set as opposed to the whole application.

Caution is required when this build flow is applied in mixed critical systems. ISO 26262 specifies that freedom from interference between software elements is required. This can be achieved by grouping the modules into sets where each set contains modules with the same Automotive Security Integrity Level (ASIL) assigned. All modules of a set are passed to the compiler in one execution. This results in one assembly/object module created per ASIL.

These object modules, in addition to libraries, are then passed to the linker which creates the final object or hex file. In multi-core systems, code and/or data can be associated with a specific core. TASKING compilers are aware of these core associations and will not create any inter-core interactions.

MIL-Splitting

When MIL-splitting is enabled, the C compiler will first link the application at the MIL level as described in the section above. However, after rerunning the optimizations, the MIL code is not passed on to the backend. The frontend will write an .ms file for each input file or library. This file type will have the same format as a .mil file.

Using this approach provides the advantage of making the tool translate only those parts of the application to a .src file that have been modified. The MIL-splitting build process is faster and more efficient that than the MIL-linking build process. However, the tradeoff to this speed and efficiency is code compaction optimization in the backend operating on a module level rather than an application level. Similar to the MIL linking, it is required for the MIL-splitting build process to link with normal libraries in order to build an ELF file as shown in the figure below.

The same guidelines should be taken into account that were described for MIL-linking when this MIL-splitting build flow is used to create a mixed critical system.

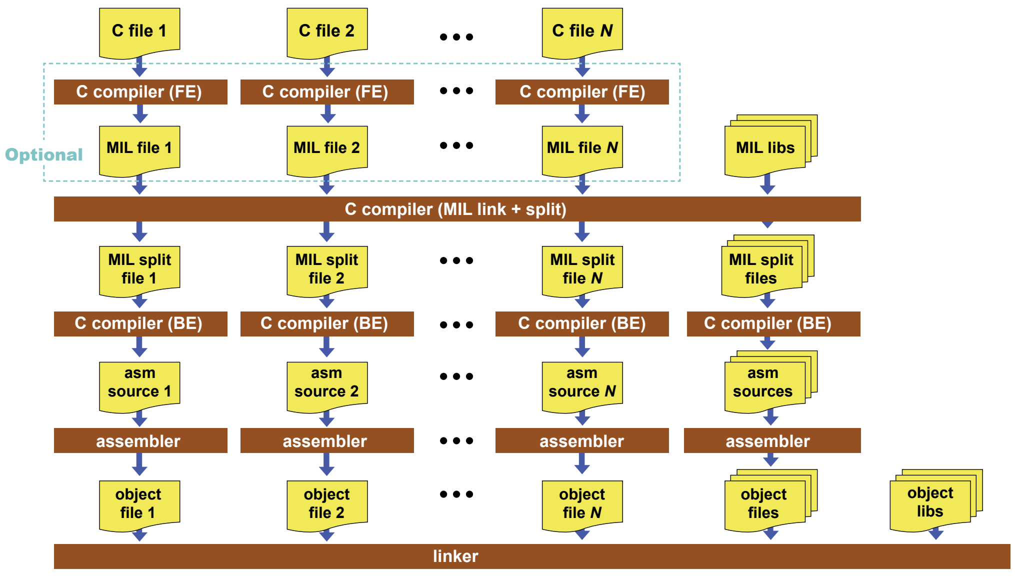

Figure 2: MIL-split build flow

Inlining

When inlining is enabled, the C compiler automatically inlines small functions in order to reduce execution time cycle by replacing the function calls with a copy of the code. The C compiler then decides which functions will be inlined. This process can be overruled with the two keywords `inline` (ISO-C) and `__noinline`. With these inline directives you can request the compiler to inline the specified function regardless of the compiler's optimization strategy. If a function with the keyword inline is not called at all then the compiler does not generate any code for it.

Inline functions should be defined in the same source module where you call the function, since the compiler only inlines a function in the module that contains the function definition. When you need to call the inline function from several source modules you must include the inline function definition in each module (e.g., using a header file). When inlining is enabled along with MIL-linking or MIL-splitting, the compiler can inline functions from any module that has been passed to the compiler. In general, inlining is considered as a speed optimization that increases a memory footprint. However, if inlining is applied before code compaction then code compaction results often increase.

Code Compaction

Code compaction works in an opposite way to inlining code optimization. This process takes chunks of source code, as well as sequences of compiler generated machine instructions that occur more than once, and replaces them by a function call that consists of the source code and sequences. The effect of this optimization technique is opposite of that of inlining as it yields a smaller code size at expense of slower execution speeds.

The code size reduction that can be achieved depends on the number of functional clones in the source code as well as on the characteristics of the processor's instruction set architecture. The most influential characteristic is the function call – return overhead, and therefore the shorter the instructions sequences, the smaller the function call it replaces. Combining inlining followed by reverse inlining can be a great optimization method when you're working with a resource constrained target or design.

When function inlining and code compaction are both implemented, the call structure of the source code will significantly differ from the call structure of the binary code. In this scenario, the compiler generates additional debug information to facilitate the debugger to reconstruct a C-level stack trace that corresponds with the structure of the original source code, allowing for the symbolic debugger to remain available.

Linker Optimizations

This linker optimization tool tackles duplicates for removal and operates at the ELF (Executable and Linkable Format) section scope, which corresponds to individual C-level functions and variables in TASKING toolsets.
If identical code/data sections are found in the object files then the sections are overlaid, and as a result the duplicate sections are removed from the FLASH image. These linker optimizations do not take the section's ASIL level into account and are therefore not suited for use in mixed critical systems.

CONCLUSION

TASKING compiler toolsets offer a large variety of build flows and optimization strategies to reduce the memory footprint of your application. Code size reducing optimization strategies can also have a positive effect on the execution speed of your application. These toolsets are highly configurable and can be customized to your exact application needs, enabling you to strike the perfect balance between build time speed and code optimization. Requirements imposed by safety standards have also been taken into consideration when designing this toolset, and advanced optimizations can be applied in safety critical software.

This technical paper was just a brief overview of some of the optimization capabilities in TASKING toolsets. Get started with your TASKING toolset today by registering for a free trial.