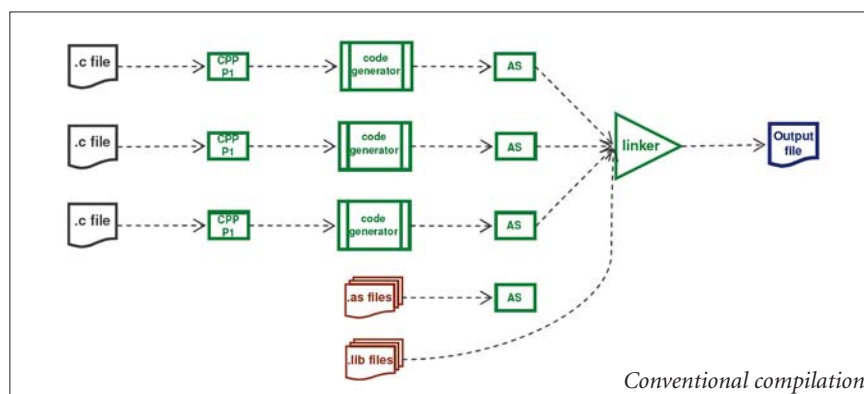


Whole-program compiler technology for increased code density

By Clyde Stubbs, Hi-Tech Software

Omniscient Code Generation, a new compiler technology, employs optimisation techniques based on a global view of the complete program that solve most of the problems associated with independent compilation.



■ The somewhat irregular architectures of embedded microcontrollers are often an awkward fit with the standard C language, frequently requiring substantial architecture-specific handcrafting to achieve efficient code. Omniscient Code Generation simplifies and streamlines the programmer's job by abstracting and hiding the underlying architecture, while simultaneously delivering reduced code size and increased execution speed.

C-language programs are usually split into multiple small modules, each of which is compiled independently, with little regard for cross-module register optimisation, re-entrant code or variable declarations. A linker hooks all these compiled program modules together, along with code extracted from pre-compiled libraries to create the final program. Conventional C-compilers are not able to take advantage of the wide variety of memory maps, and register and stack configurations available in today's microcontrollers. All too often, the programmer must resort to manual optimisations that compromise portability. In addition, compiling program modules independently can introduce several types of problem into the program because each module is compiled without knowledge of the other modules. Today there is no reason to compile each module independently. There are many reasons not to do so. Variables and other objects may not have consistent

definitions across all program modules. Conventional compilers have no way to check for this problem prior to object code generation. And, although it is possible to have the linker check for incompatible re-declarations of variables by different modules, this approach adds complexity and does not always solve the problem due to the fact that the linker may not have enough information to detect the human error. The way in which arguments are passed and values returned, is another potential drawback in independent module compilation. Each calling convention contains a set of rules that defines which CPU registers are to be preserved across calls. All functions must adhere to the same calling conventions. Since it is impossible to know at the time of compilation which registers will and will not actually be used by a called function, these rules can result in sub-optimal register allocation. This is particularly true with small embedded processors.

Processor memory architectures pose additional problems. Many embedded processors have a complex and non-linear set of memory spaces, often with different address widths. Standard C, which assumes a single linear address space, is difficult to map onto these architectures because it is often impossible to know at compile time which memory space a variable will be located in or what spaces a given pointer will be required to access. For example,

a pointer might need to address memories of different address widths, such as an 8-bit wide RAM and a 16-bit or wider ROM. The programmer can achieve efficient pointer usage in these architectures by explicitly declaring the address spaces that a pointer will access. However, this solution is architecture-specific and results in non-portable code. It also increases the likelihood of subtle bugs being introduced.

Many small embedded processors do not have a fully or efficiently addressable stack for the storage of local variables. This situation is usually handled by a compiled stack where local variables are statically allocated in memory, based on a call graph. Unfortunately a conventional compiler does not know the call graph until link time. Using a compiled stack requires the programmer to nominate any functions that are called re-entrantly, so that they can be dealt with accordingly. If the programmer nominates incorrectly, the error is not revealed until link time, by which time it is too late to make changes.

Today's fast, memory-rich desktop processors are able to generate code for the entire program at once, allowing all routines, variables, stacks and registers to be optimised based on the entire program. Although it is theoretically possible to compile a very large program from a single source, the fact is that most programs are

written by teams of engineers, each of whom is responsible for one or more modules of functionality. As a result, embedded systems will always be compiled from multiple files. A new compiler technology, called Omniscient Code Generation (OCG), takes into account all the variables, functions, stacks and registers represented in all program modules. Rather than relying completely on the linker to uncover errors in independently compiled modules, an OCG compiler defers object code generation until a view of the whole program is available. It employs optimization techniques based on this global view of the complete program that solve most of the problems associated with independent compilation. Instead of compiling each program module to machine code instructions in an object file, it compiles each module to an intermediate code file that represents a more abstract view of each module. However, it does not produce actual machine instructions or allocate registers at this time. Once all the intermediate code files are available, they are loaded by the code generator into a call graph structure.

The code generator also searches intermediate code libraries and extracts, as required, any library functions that are referenced by the program. Once the call graph is complete, functions that are never called may be removed, thus shrinking the total amount of code. The call graph also allows identification of any functions called recursively or re-entrantly, (e.g. those called from both main-line code and interrupt functions). These functions must either use dynamic stack space for storage of local variables, or be managed in other ways to ensure that a re-entrant call of the function does not overwrite existing data. This is achieved in a way that is completely transparent to the programmer and without using any non-standard extensions to the language.

If a compiled stack is to be used, it can be allocated at this point, before any machine code has been generated. OCG knows exactly how big the stack is and where it is located, before the code is generated. The compiler then builds reference graphs for every object and pointer in

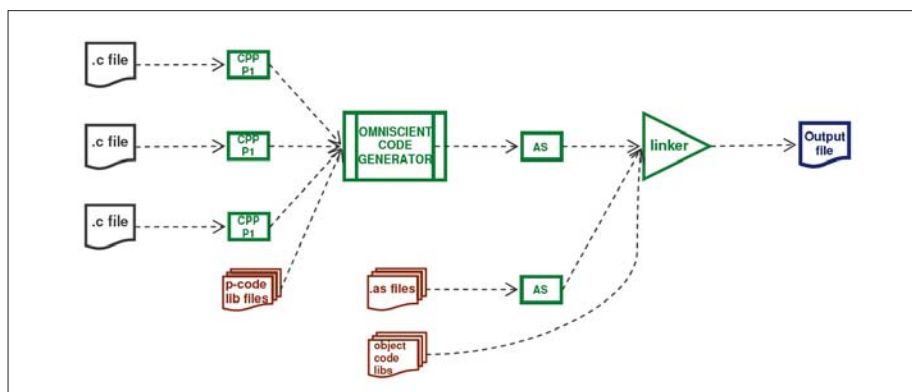
the program. Any conflicting declarations of the same object from different modules can be detected at this point and an informative error message issued to the user. Any variables never referenced can be deleted.

Determining the memory space for each pointer is one of the most important features of OCG. An algorithm uses each instance of a variable having its address taken, plus each instance of an assignment of a pointer value to a pointer (either directly, via function return, function parameter passing, or indirectly via another pointer) and builds a data or pointer reference graph that identifies all objects that

can possibly be referenced by each pointer. This information is used to determine which memory space each pointer will be required to access. Once the set of used variables and pointers is complete, OCG allocates memory of both the stack (compiled or dynamic) based on the call graph and allocates global and static variables based on the pointer reference graph. Where there are multiple memory spaces (e.g. an architecture with banked RAM), the variables accessed most often in the program can be allocated to the memory spaces that are cheapest to access. On an 8051, for example, this would be internal, directly addressable RAM, rather than external RAM which must be

```
Machine type is 18F452
Call graph:
*_main size 0,6 offset 0
*_dummy size 0,5 offset 6
*_fcp size 2,12 offset 11
  awtoft size 0,0 offset 11
  _free size 0,2 offset 6
*_another_isr size 0,0 offset 25
*_my_isr size 0,6 offset 25
  lbtoft size 0,0 offset 31
*_delay size 2,0 offset 31
```

Call graph



Omniscient code generation

accessed via a pointer. Each pointer variable now has a set of address spaces that it will be required to access. This allows each pointer to be sized and encoded in a way which is optimally efficient for the particular architecture, while still being accurate. This process is entirely automated and requires no specific directions from the programmer. Generation of machine code begins at the bottom of the call graph, starting with those functions that do not call any other functions. Automatic in-lining of these functions may be performed if desired. In any case, the code can be generated without the constraints of rigid calling conventions. Code generation then proceeds up the call graph, so that for each function, the code generator knows exactly which functions are called by the current function, and therefore also knows exactly what registers and other resources are available at each point. Calling conventions can be tailored to the register usage and argument type of a function, instead of following a set pattern.

Since an omniscient code generator has a truly global view of the program, it can customise complex C library functions for each program. For example, C library functions `printf()` and `fprintf()`, used for formatting text strings or output can occupy 5 kilobytes or more if imple-

mented in their entirety. The OCG code generator can analyse the format strings supplied to these functions, and determine exactly the set of format specifiers and modifiers actually used by the program. This information can be used to create a customized version of the function that contains only the specifiers and modifiers required by the program, with potentially enormous reductions in code size. Code for a minimal version of `printf()` implementing simple string copying can be as little as 20 or 30 bytes, whereas a version providing real number formats with specific numbers of digits could occupy 5000 bytes or more. No programmer input, other than writing the program itself, is required to benefit from this customization and optimisation.

Many library functions return values that are not necessarily checked by the calling code. A compiler with OCG can establish whether the return value of a function is ever used. If the return value for a particular function is never used, the OCG compiler removes the code that implements the return value in that function. A “conventional stack” is implemented in the hardware of the target MCU. However, not all MCUs have a hardware stack, so the compiler must implement a “compiled stack” built at compile time rather than runtime. Although

this is not a great problem for most embedded applications, it makes writing re-entrant code difficult. A compiled stack is not as good as a hardware stack as it cannot implement recursion or re-entrant function calls. OCG gets around re-entrant function calls by building separate call graphs for both main-line and interrupt code. Any functions that appear in more than one call graph can be replicated, each with its own local variable area. Using this technique re-entrancy can be implemented without a conventional stack.

The C language requires that uninitialized static and global variables be cleared to zero on startup. Many newer embedded compilers provide canned startup code for this purpose. However, canned startup code is often much larger than necessary for a given program. For example, if the program has no uninitialized global variables, there is no need to include code to clear them. Since the omniscient code generator can make this information available to the code generator, it can create the smallest possible runtime startup code. In a minimal case, the startup code may be completely empty.

One obvious advantage of OCG is smaller, faster code. More importantly, OCG allows embedded C programs to be written without the use of architecture-specific extensions. By performing at compile time an analysis of the whole program, the code generator can make the decisions about memory placement, pointer scoping etc. that would otherwise be made by the programmer using special directives or language extensions. This analysis is performed every time the program is recompiled, so it is always accurate and up-to-date. OCG compilation technology results in substantially denser code and concomitantly, faster program execution. For example, a C-language source file compiled for Microchip’s PIC18 using High-Tech Software’s PICC-18 PRO was 49% smaller than the code generated by IAR’s EW18 compiler. ■