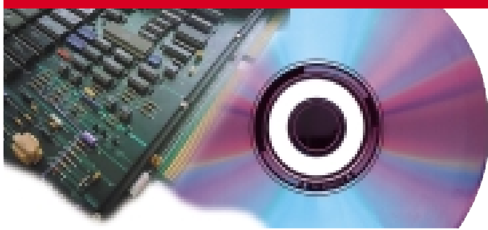


Tools

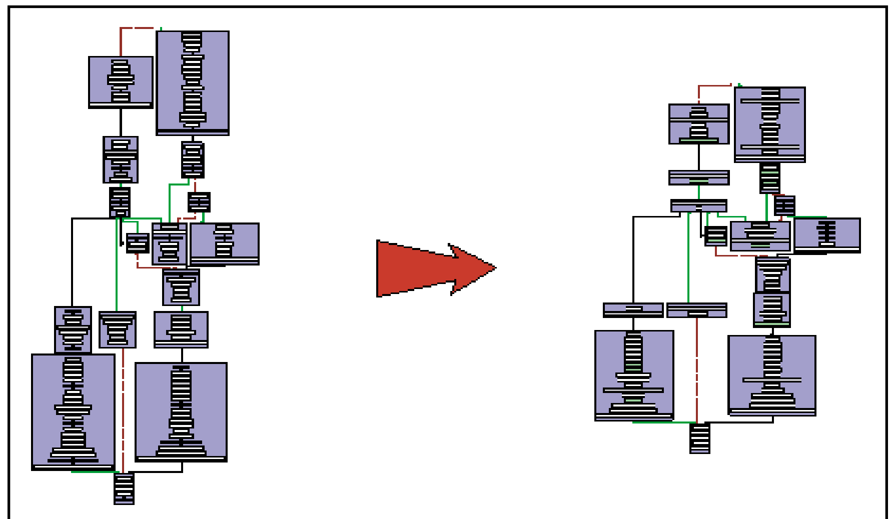
Post Pass Code Compaction at the Assembly Level for C16x

by Christian Ferdinand

AbsInt Angewandte Informatik GmbH
http://www.AbsInt.com
ferdinand@AbsInt.com

aiPop166 post pass optimizer suite was developed to reduce code size and improve the code quality of assembly files produced by Tasking's C compiler for C16x.

The size of compiled C code is becoming increasingly important in embedded systems, where the economic incentives to reduce ROM sizes are often very compelling. By combining advanced static program analysis methods and pattern matching techniques it is possible to reduce the code size of programs, while preserving the ability to run the program executable directly, i.e. without an intervening decompression stage.



In recent years there has been an increasing trend towards using microcontrollers in a wide variety of consumer and industrial products. However, the amount of available memory is limited due to price, space, weight, power consumption etc. At the same time, there is also a growing desire for enhanced functionality in these products, this implying increasing amounts of code. Unfortunately, slight increases in code size can lead to high additional costs, e.g. when having to switch from on-chip memory to off-chip memory or add additional flash memory (currently expensive). In the worst case considerable effort and expense might have to be invested in migrating to another microcontroller model. Tools for reducing code size help to save considerable time and money. This is where the aiPop166 post pass optimizer suite comes in, as it was developed to reduce code size and improve the code quality of assembly files produced by Tasking's C compiler for C16x.

An optimizing compiler usually works like this: The input program is read, checked for syntax errors, and transformed into an intermediate representation (IR). At the IR level a set each of target-architecture-dependent and -independent optimizations is performed. Code selection, register allocation and, depending on the target architecture, instruction scheduling are the tasks of the compiler back-end transforming the IR into assembly code. The code quality and code density of modern optimizing C compilers on regular architectures is usually quite high. Nevertheless, by using a post pass approach that works on the resulting assembly files, it is often possible to reduce code size considerably. There are basically two reasons why a post pass optimizer is able to achieve this:

1. Larger scope: For complexity reasons, compilers can only consider (and optimize) a small part of an input program at a time. Yet in a post pass approach the scope can be much larger, e.g. an entire procedure, module or even the entire application.
2. Lower level: Most compiler optimizations work at the IR level. However, additional gains can be made by reapplying the same optimizations to the lower assembly level, this applying in particular to target-architecture-dependent optimizations.

The following sections provide details of the optimizations performed by aiPop166.

Functional Abstraction and Tail Merging

The underlying idea of functional abstraction and tail merging is to:

1. identify multiple occurrences of instruction sequences,

2. make one representative sequence that can be used in place of all the other occurrences, and

3. have the optimized program use the representative instead of the occurrence. This optimization is referred to as *functional abstraction* if it is achieved by a function call or *tail merging* if it is achieved by a jump from one procedure into another.

Example

Consider the following code sequences:

```
 CALL _strcpy_x
MOV R12,#06h
ADD R12,R0
AND R12,#03FFFh
MOV R13,DPP1
MOV R14,#POF _10
...
MOV [R0+#04H],R6
MOV R12,#06h
ADD R12,R0
AND R12,#03FFFh
MOV R13,DPP1
MOV R14,#025h
```

The repeated code sequence can be extracted into a procedure (functional abstraction). This transformation reduces the code size:

```
 _aipop166_bb_43f PROC FAR
    MOV R12,#06h
    ADD R12,R0
    AND R12,#03FFFh
    MOV R13,DPP1
    RETS

CALL _strcpy_x
CALL _aipop166_bb_43f
MOV R14,#POF _10
...
MOV [R0+#04H],R6
CALL _aipop166_bb_43f
MOV R14,#025h
```

Interprocedural Constant Propagation at the Assembly Level

For many microcontrollers using large constant values (e.g. absolute addresses of variables or code, bit patterns, ...) inflates code size. In the C16x, for example, access to memory via indirect addressing using a register requires a 2-byte instruction whereas using the absolute address requires a 4-byte instruction. Consequently, 2 bytes can be saved by replacing a move instruction containing an immediate value that has already been loaded into a register by a register-to-register move instruction.

Optimizations Based on Data Dependency Analysis

Due to the semantics of C, a compiler often generates instructions whose result is never used or it generates instructions for loading data that has already been loaded. In addition,statements whose results are never used are to be found at the C program level as well. However, it is usually quite hard for a compiler to identify such instructions due to its more local view. aiPop166 employs a procedure-wide data dependency analysis to remove such unnecessary instructions.

EXTP Optimizations for FAR Data

Access to FAR data on the C16x is quite expensive due to its 16-bit architecture. When an application is migrated from small memory model to large memory model code is usually observed to grow considerably. Each access to FAR data usually requires an additional instruction to set a DPP register to the correct page address or an EXTP instruction, this having an effect similar to setting a DPP register locally.

aiPop166 uses heuristics based on two interprocedural analyses that try to change the settings of DPP registers and EXTP instructions so that a DPP value can be reused.

Peephole Optimizations

Apart from the optimizations mentioned above, aiPop166 includes a set of simple optimizations to reduce code size and improve execution speed, e.g. removing unused procedures, empty procedures, unnecessary NOP instructions and zero-displacement jumps, in addition to performing tail-call optimizations.

Speed vs. Size

Functional abstraction and tail merging result in a runtime penalty due to the additional CALLS/RETS or JUMP instructions. aiPop166 enables a tradeoff between speed and size. By sacrificing the compaction rate very slightly the overall runtime overhead can be kept low by only extracting code sequences of a given minimum size. Using pragmas enables timing-critical parts of the application to be excluded from any optimization that might result in slower execution or from any optimization at all.

Integration in the Development Cycle

Functional abstraction and tail merging work best when applied to the entire application. However, applying global optimization for each step in the development process (compile-test-debug-change) can be very time-consuming. aiPop166 enables complete reanalysis of the application code to be avoided as it takes advantage of the circumstance that an application usually changes only very moderately over time. During the learning phase, aiPop166 collects information from the entire application code on repeated code sequences worth extracting. On small applications (say 64 KB of code segment size) this may take a few minutes, whereas on large applications (> 1 MB of code segment size) this may take a few hours. This information is stored in a database file. This database file is later used during the optimization phase to extract repeated occurrences of code sequences. Using this pre-computed information enables the scope of the optimization phase to be restricted to one module (.c file). Result: The optimization phase is fast, thus enabling invocation of aiPop166 to be integrated in the makefile controlling compilation of the application.

The compaction rate drops slightly over time as the application code changes. When the application code has changed considerably and the compaction rate has dropped under an acceptable limit, the learning phase can be restarted. As an alternative, the learning phase can be invoked regularly on weekends, at night or during lunchbreaks.

Debugging

aiPop166 leaves most of the symbolic debug information intact. Nevertheless, this information has to be omitted for extracted program parts. Since the extracted program parts are usually only a few instructions long, source-level debugging is influenced only slightly. In addition, extraction optimization can be temporarily switched off for the program parts in question.

Compaction Rates

aiPop166 has been tested with several real applications, with size reductions over 66% having been observed for specific modules. Testing of entire reference customer applications has shown an overall size reduction of between 4.8% (small application featuring highly hand-optimized C code) and 20.39% (large mobile phone application). A reduction of 20% means that 25% more code and functionality can be packed into a flash memory of the same size.

Supported Compilers and Platforms

aiPop166 optimizes assembly files in .src format produced from regular C files by Tasking's C compiler for C16x. It is available for Solaris, Linux and Windows NT.

❑