# Retargeting GCC Compiler for Specific Embedded System on Chips

BENBIN CHEN[1,2], XIAOCHAO LI[1], DONGHUI GUO[1,3]
[1]Department of Electronic Engineering
Xiamen University
Xiamen Fujian 361005, China
[2]School of Electrical Engineering and Automation
Xiamen University of Technology
Xiamen Fujian 361024, China
[3]IC Design & IT Research Center of Fujian Province
Xiamen, China
chenbenbin@163.com, leexcjeffrey@xmu.edu.cn, dhguo@xmu.edu.cn

*Abstract:* - This paper describes the High-performance C Compiler (HCC) and its specific implementation for industrial application-specific embedded System on Chips (SoCs). HCC compiler is a language C compiler based on the retargetable GCC compiler. Because of the specialized architectures and features of specific embedded SoCs, machine-dependent compiler implementation is an important and challenging work in embedded system. To quickly implement a compiler for specific embedded SoCs, compiler extension methods are proposed in HCC compiler. We extend the identifier and attribute with Abstract Syntax Tree (AST) for language-specific programming syntax of the compiler front-end, which is the syntax of the extended standard ANSI C. And then, the machine-dependent classification of assembly generation for the specific embedded SoCs is designed and implemented. After finishing the ABI (Application Binary Interface) and MD (Machine Description) of the compiler back-end, the HCC compiler is completed by retargeting GCC compiler for the application-specific embedded SoCs. These implementation methods could be referenced for other embedded chips. According to the crossing contrasts and tests with multiple compilers of the same type, conclusion can be drawn that the proposed HCC compiler has a stable performance with excellent improvement of the generated assembly codes.

*Key-Words:* - Compiler; Specific Embedded SoCs; Language-Specific; Machine-dependent; Abstract Syntax Tree; Attributes;

## 1 Introduction

Application-Specific Instruction-set Processor (ASIP) is quite common in embedded system design [1]. The embedded SoCs based on the ASIPs are more complexity and diversely existed in the semiconductor market, which have to meet very high efficiency requirements for high quality optimizing compiler. Compared with the compiler of general processor, the compilers of specialized architectures of embedded SoCs based on ASIPs are often insufficient in fully exploiting the processor capabilities demands via more dedicated code and optimization techniques [2]. The effectiveness of compiler implementation for embedded SoCs is also determined by the combination of target architectures, target application, and the compilation environment [3], [4]. GCC (GNU Compiler Collection） [5], [6], [7], [12] which can support multiple programming languages in the front-end and more than 30 processors in the back-end, is the most widely used compiler collection. Because of its excellent flexibility and retargetability, it can be often seen in the transplant of the cross compiler as retargeting compiler. However, thought GCC is a robust and well-supported compiler, which can be retargeted by means of a Machine Description (MD) files that captures the compiler view of a target processor, it is very complex, hard and generally results in huge retargeting effort and it needs the effective improvement to customize and change the compiler back-end and front-end for the irregular architectures. For example, the DSP processors often have the irregular registers set which need additional modifications of compiler back-end to adapt the new situation [1]. The specific embedded SoCs used in this paper are the harvard architecture microcontroller with own instruction-set which is very similar with the DSP processors, so it is urgently required the high performance C Compiler for programmers to advance the development

efficiency for more and more complicated embedded software applications with high-level C language.

Based on their own purposes, there are many previous studies to discuss the GCC transplants. The construction specification of a compiler front-end is described in [9], [10] and [11], which demonstrate how to design appropriate front-end language for specific fields and applications. Reference [2] and [11] describe how to add a support to the bit data type by following the processing hardware mechanisms. Reference [12] and [13] implements the transplantation of the GCC to the hardware platform with different architecture, for example, the CRIS and VLIW processors. A discussion on the optimization of the compiler for low power consumption and multi-core parallel processing is made in [18] and [19]. Based on intensive studies of the processing mechanisms of the compiler, reference [2], [7], [8], [12], [13] and [16] introduce the back-end design and carries out the transplantation to specific processors. [14], [15], [20] and [21] extend the application fields of compiler to machine learning, domain-specific semantics for various kinds of programming environments and applications. However, the compiler presented in these references mainly focuses on the back-end design, such as the design of the MD (Machine Description), or the single extension of the front-end [2], [17], it is not proposed the general extension methods of GCC compiler and it is natural to observe the difficulties in the transplantation and development of the complier for embedded SoCs based on GCC [1].

Here in this paper, a compiler extension methods aiming at expanding the front-end and back-end of the GCC compiler is proposed. The methods, which combines the commercial specific embedded SoCs and the programming syntax extension of standard ANSI C, achieves not only the extension and limitation of the identifier in the compiler front-end, but the implementation of the MD and ABI for machine-dependent design to produce the suitable assembly codes in the compiler back-end. More importantly, we propose a new attribute of the syntax tree for the language-specific programming syntax and combine the new attribute to the already existing AST tree so that compiler back-end could use it to produce the correct assembly codes. The High-performance C Compiler (HCC) with the extension methods is completed and comparisons have been made. The experiment results have shown the excellent performance of the proposed methods in meeting the high demands of the embedded SoCs. It is very useful for the compiler porting and implementation of various other kinds of application-specific embedded processors which are in urgent need of powerful, flexible and stable compiler.

## 2 The Specific Embedded SoCs Architecture

The Specific Embedded SoCs used in this paper are Harvard architecture processor that saves its instructions and data separately. The commercial embedded SoC is based on the RISC instruction set and adopts a paged saving mapping, independent One-Time Programmable (OTP) program memory, data memory, stack and bus, so it is able to visit the program and data at the same time. The Fig. 1 is the typical diagram for embedded SoCs which processor could be called HCC processor. They are many optional peripheral controllers that affect the layout of interrupt vector table.

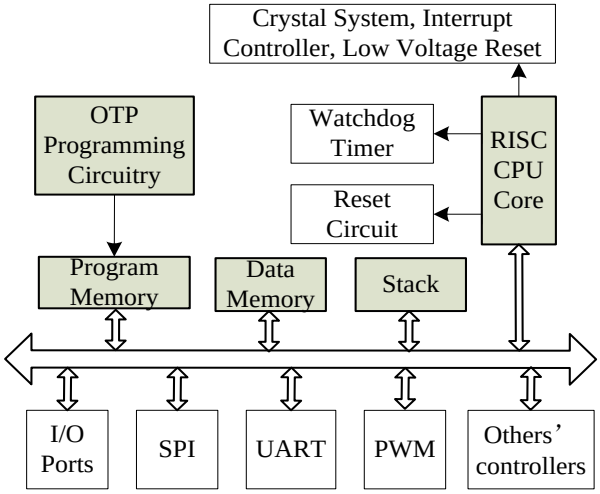



**Fig. 1** The block diagram of embedded SoCs

The HCC processor also leaves out the most internal general registers and uses the specific section of the data memory or the specified address instead. The special memory space and the multi-paged saving require the compiling syntax to be able to manage the sections of the internal storage flexibly and to have a strong protection mechanism, to avoid the illegal operation of the internal storage, which make the design of the complier more difficult with correctly implementation.

The program memory in Fig. 2 is the location where the user codes or programs are stored. OTP devices offer users the flexibility to freely develop their applications which may be useful during debug or for products requiring frequent upgrades or program changes. The program memory is subdivided into several individual banks each of 8K

capacity. Within the program memory, certain locations are reserved for special usage such as reset and interrupts. The program memory bank is selected using the bank pointer, which is also used to control the data memory bank pointer. The data memory showed in Fig. 2 is a volatile area of 8-bit wide RAM internal memory and is the location where temporary information is stored. Divided into two sections, the first section of these is an area of RAM where special function registers (SFR) are located with fixed locations. The second area of data memory is reserved for general purpose use, which is divided into several separate banks, known as bank 1~bank n.

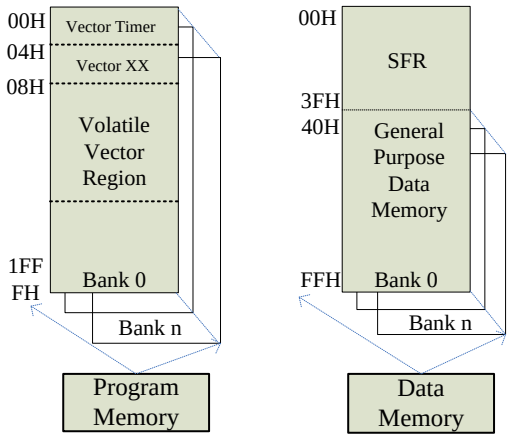



**Fig. 2** The program and data memory structure

## 3 The Extension Methods of HCC Complier

The compiler is carried out in stages while each stage is in charge of transferring the source program from one to another. The retargetable GCC mainly consists of three modules namely, the front-end, the middle layer and the back-end showed in Fig. 3. The GCC front-end is in charge of the preprocessing, lexical analysis, grammatical analysis, generating the corresponding abstract syntax tree (AST) and eventually generating the generic tree which is the unified tree structure for GCC all kinds of language front-end such as C, Java, C++, Ada, Fortran and so on. The middle layer is responsible for the transformation and optimization of the syntax tree, including the transformation of the gimple tree, the SSA (Static Single Assignment) transformation, the N-pass optimization based on SSA transformation, and generating the RTL (Register Transfer Language) intermediate representation, which is not related to the target platforms (e.g. the ARM processor, the MIPS processor etc.) but closes to the final syntax tree in the form of assembly codes and will generate the object assembly codes in combination with the back-end MD template.

GCC compiler uses the machine description method in back-end to complete the retargeting requirement. As a retargeting compiler, it can use machine description in back-end including the MD file (e.g. target.md), ABI (e.g. target.h) and auxiliary file (e.g. target.c) for specified target processor to retargeting a compiler of particular processor. For an unknown processor with own instruction set (e.g. the embedded SoCs used in this paper), we need to finish the whole MD file and ABI file of compiler back-end.

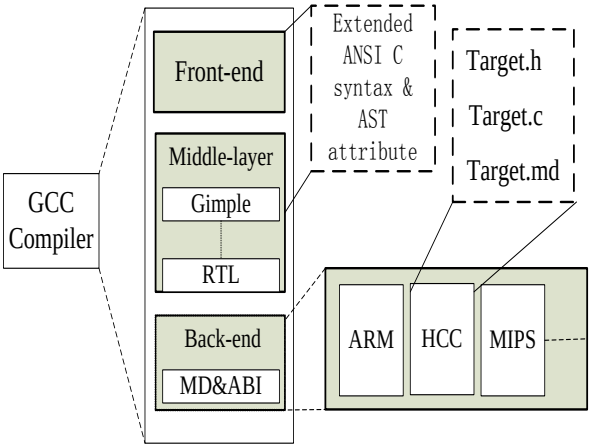



**Fig. 3** Extension and implementation descriptions of HCC compiler

The extension methods of the HCC complier proposed in this paper is shown in Fig. 3. As shown in the picture, according to the character of the architecture of the specific embedded SoCs and the extension grammar of the ANSI C syntax, the proposed HCC complier is able to carry out the extension of the lexical analysis of the special identifier for the extension programming syntax used in HCC front-end. For adding the new attribute, the new attribute keyword '@' is parsed. HCC compiler creates the new attribute nodes and combines the new attribute to the existing AST as special explanation of new programming syntax. The machine-dependent implementation with MD file and ABI file are completed to produce corresponding assembly output according to the instruction set of the embedded SoCs. Besides, the front-end and back-end of the proposed HCC complier are thoroughly detailed designed based on the syntax rules and the architecture of the embedded SoCs which it is going to support.

# 4 Language-Specific Implementation

Using GCC intermediate representation, HCC also adopts the AST structure to describe the language front-end. The basic structure of AST is the struct *tree_node*, which could be represented as a data type for a variable, an expression, even a statement in each tree node [8]. On behalf of the object, the enumeration type constant *tree_code* is used in every *tree_node* to express the types of nodes. Typically, INTEGER_TYPE represents integer type, ARRAY_TYPE represents array type, VAR_DECL is for variable and FUNCTION_DECL is for functions and so on. In the AST representation, all the information put in its code by the programmers could be covered, everything concerning the control flow, everything about structures and types [5], [6]. As showed in Fig. 4, generic, gimple and SSA are also simple form of AST which records the language-independent information for optimization.

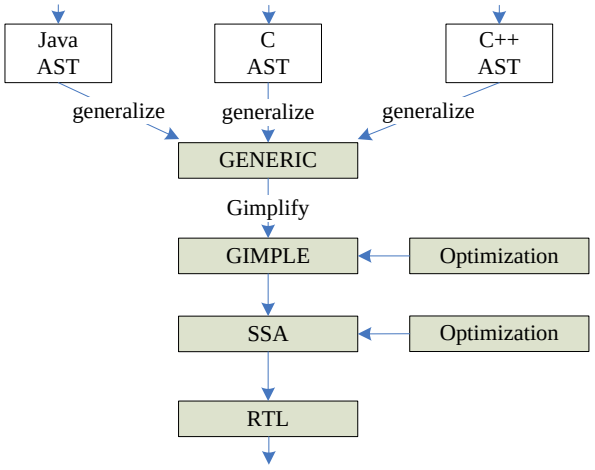



**Fig. 4** AST intermediate representation

GCC attribute is similar to a kind of context, which also is in the form of nodes in AST and records a wealth of semantic information. there are three types of attributes, respectively are variable attributes, type attributes for structure, such as union and enumerated types and function attribute which is attributes applying to functions. Each attribute is one of the following formats showed in Table 1.

GCC itself has a large number of variable, type and function attributes, which have their own expressions and the processing functions. Combined with the mechanism of attribute, this paper extends and customizes the inserted attribute for AST expression of HCC language-specific programming syntax, which extends the language-specific syntax of front-end. Meanwhile, the automatic combination of the special attribute of AST to given object (function, variable or type) is implemented. Then the attribute could be passed forward to target back-end for assembly codes generation.

**Table 1**. Example of the attribute formats.

| Formats | Example |
|---|---|
| Empty | __attribute__ (()) |
| A word | __attribute__ ((unused)) |
| A word with parameters | __attribute__ ((format_arg (2)) __attribute__ ((format (printf, 2, 3)) |

When extending the language-specific syntax, the callback functions and hooks are used in this paper. These interfaces are the important interactive mode for language related front-end or the interaction between the front-end and back-end, which are frequently used in compiler implement in porting or optimization. Taking advantage of the way of the program callback, combining with the GCC attribute, a controlled extension mechanism for target-specific C compiler is addressed in this section.

## 4.1 The extension of the identifier and SFRs

In order to express the special variable, type in memory and the location with address made by function, the HCC would need to add the character '@' as the keyword to extend the front-end syntax. The grammatical features of '@' is as follows.

*Syntax:*

*Data_type Variable|Type|Function @ memory_location*

This is the HCC specific description way for variable, type, and function with corresponding saving address. Firstly, the character '@' must be recognized as the special identifier in the function *lex_identifier* in GCC sources codes during the lexical analysis stage. Finally, it needs to be recognized as the keyword in the function *c_lex_one_token*.

Since the identifier I in C language has following syntax rules:

L → A | B | … | Z | a | b | … | z |;

D → 0 | 1 | … | 9;

I → L | _ | ID;

Adding the '@' will break the definition of the original identifier. Considering that the GCC carries out the lexical analysis via the *lex_identifier* function (shown in Fig. 5), certain modifications have been made to it to avoid the break caused by '@' and guarantee it can be recognized as the identifier. '@' in HCC is similar to objective C,

which need the '@' as <CPP_ATSIGN> in the token scanner, but in objective C it doesn't take pass to the parser as a grammar element such as GCC attribute in the compiler.

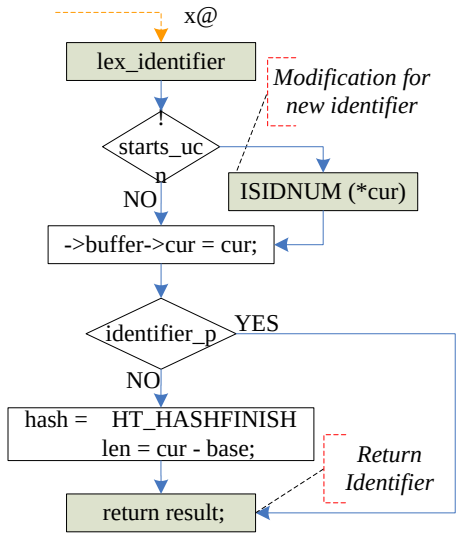



**Fig. 5** HCC identifier transformation

The SFRs in the application-specific embedded SoC adopt a mode of specified data address in data memory, which means the distribution of the data storage has already been known. An example of SFR is given as follow:

*//define a SFR register "_reg0"*
*volatile int reg0 @ 0x00*

The character '@' is used in programming and compiling stages of HCC compiler to set the address of the SFRs. Meanwhile, it is also allowed to set the address for the variables defined by the users with '@'. Combining the handling '@' as attribute keyword <RID_ATRRIBUTE>, we can identify the attribute keyword and parse the corresponding legality of SFR address. And then, the SFR address (*0x00* in above example) could be added to variable AST as a special variable attribute.

## 4.2 Pragma and the control of functions

The embedded SoCs adopt the banks storage structure, in which a great deal of p*ragma* control grammatical features is designed. For example, the specific attribute of both the normal functions and interrupt functions can be controlled by *pragma*. The *pragma* control syntax of the interrupt function in HCC compiler is as follows:

*#pragma vector symbol @ address //interrupt indication with reserved word "vector"*
*void symbol ()*
*{*
*function codes; //implementation codes of interrupt*
*}*

The above syntax has used *vector symbol* and *address* to indicate the interrupt function and the storage address. As an extension programming syntax, the syntax uses *pragma* to express the interrupt and to identify the address of the interrupt function. In combination with the attribute handling with <CPP_ATSIGN> in HCC compiler, this paper proposes a method of constructing an attribute tree chain of the functions or interrupt functions to manage the attribute of the normal functions and interrupt functions with addresses as shown in Fig. 6.

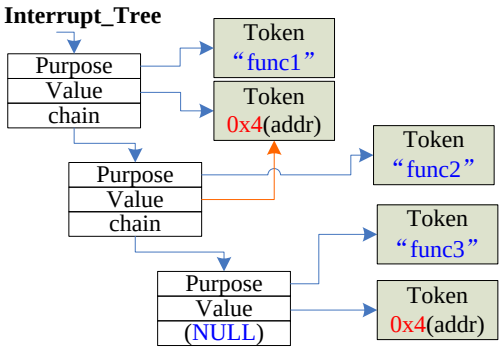



**Fig. 6** Tree chain based on interrupt attributes

From the Fig. 6, there are two key fields in the nodes of tree chain. The first one is the purpose field which is used to record the function name and another one is used to save the function address. For interrupt functions, tree chain is used to link all of the vectors that are indicated by *pragma statements* with reserved word "*vector*". And then, the nodes of tree chain could be parsed as the interrupt attribute of specified function. The interrupt processing with *pragma statements* are listed below in language-specific front-end of HCC compiler.

a. register the *pragma* callback function *target_pragma_interrupt* with the reserved word *vector* statically.

b. analysis the <*CPP_ATSIGN*> as the attribute keyword in the lexical analysis like *section 4.1*.

c. analysis the callback function *target_pragma_interrupt* for *pragma vector statemests* (by *target.c*) and judge the validity of the *pragma* syntax. Then generate the corresponding attribute tree chain with function names and vector numbers showed in Fig. 6.

d. identify and insert the attribute, check the corresponding function and the tree chain. Use the process showed in Fig. 7 to combine the corresponding interrupt attribute to function AST by *target_insert_attributes*.

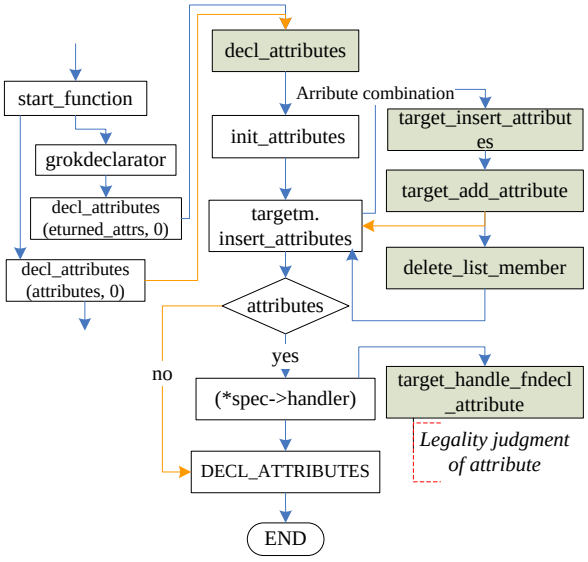



**Fig .7** Combine the corresponding interrupt attribute tree to function AST

After the combination of the interrupt attribute to the AST syntax tree is done, the compiler back-end will handle the push stark and pop stark for interrupt function according to the attribute. In Section 5.2, the new attribute handling of interrupt function will be expressed. Considering the specific embedded SoCs have multi-paged RAM/ROM with different memory configuration and lacks the general registers, the implementation and limitation modules of the target platform are added to guarantee the correctness of the assembly codes in the HCC compiler back-end.

## 5 Machine-Dependent Implementation

The compiler back-end translates intermediate representation (IR) into machine-dependent assembly codes for embedded SoCs. Compiler back-end captures the compiler-relevant machine resources, including the instruction set, register files and all kinds of requirements and constraints. In most cases, the IR represents the input source codes as assembly-like yet machine-independent low-level codes, which is the three-address code [1]. In GCC, intermediate representation called register transfer language (RTL) is an important part for the assembly codes generation. In this language, the instructions (called *insn*) are described one by one as assembly-like statements [5], [6].

RTL uses five kinds of objects: expressions, integers, wide integers, strings and vectors [6]. Expressions are the most important one. As shown in Fig. 8, An RTL expression (RTX, for short) is a C structure, but it is usually referred with a pointer (a type that is given the typedef name RTX). In the IR of the RTL, a series of optimization works related to the target machine are performed, including instruction matching and selection, instruction scheduling, register allocation, branch prediction and peephole optimizations and so on. Finally, RTL is converted into target-specific machine assembly codes. For a given IR, there is only an infinite number of mappings as well as numerous constraints exist, which is clearly a complex optimisation problem [1]. In fact, even many optimization subproblems in code generation are NP-hard [1]. As shown in Fig. 9, machine descriptions in the back-end are mainly composed of two parts, one is a C header file of machine-dependent macro definitions called target.h (*.h), which describes the ABI of target machine, and another one is a file of instruction patterns called target.md (MD file), which contains a pattern for each instruction that the target machine supports or at least each instruction that the compiler needs to be informed. In addition, machine descriptions usually contains a target.c (*.c), which is an important role for machine-independent implementation and provides the supports to files target.md and target.h with common and specific functions. i.e., guiding the generation or optimization of RTL and producing complex assembly codes.

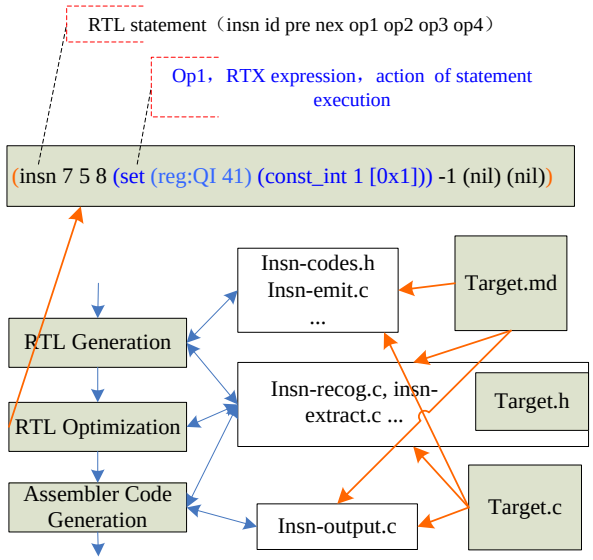



**Fig. 8** The related modules diagram of back-end

Due to the irregular architectures (lacked of registers with Harvard architecture) of the embedded SoCs, it is very complex and generally results in huge retargeting effort and need the effective improvement to customize for machine-dependent implementation of the irregular architectures in spite of the fact that GCC is a robust and well-supported compiler, which can be retargeted by means of a machine description file that captures the compiler view of a target processor in a behavioural fashion.

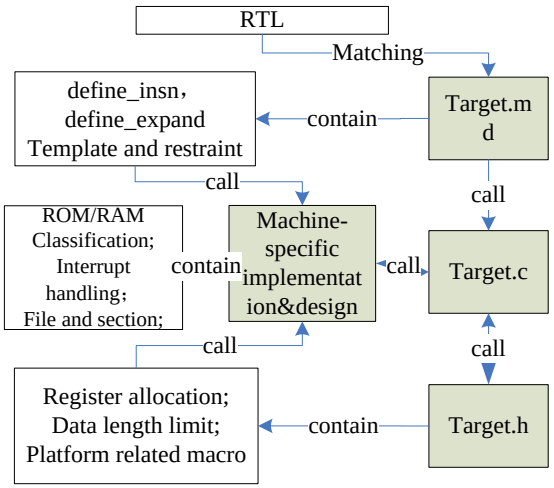



**Fig. 9** Target-specific design and implementation

As shown in Fig. 9, considering the specific platform has specific memory configuration and lacks the general registers, the machine-specific implementation and design module (e.g. for memory classification etc.) is added to process the correct assembly generation and to guarantee the quality of the assembly codes in this paper. HCC completes the machine-dependent assembly generation with the new module, MD and the ABI implementation.

## 5.1 Memory classification and MD

As shown in Fig. 8, MD file target.md takes part in two important conversions that happen in the compiler. one is process of the parse tree being used to generate an RTL insn list (PT to RTL) based on named instruction patterns of target.md, another one is the process of the insn list being matched against the RTL templates to produce assembler codes [5], [6] (RTL to ASM). File target.md defines a number of instruction patterns which are used to describe the target machine to support the operation type and assembly instructions, etc. These instruction patterns play the key roles in the conversions mentioned above, from PT to RTL and from RTL to ASM. It generally can be divided into the standard instruction patterns and non-standard instruction patterns. The formats of instruction patterns and RTL templates have been discussed in previous studies [5], [6], [7], [12], [13], which are not the emphasis here.

| Field | Example |
|---|---|
| Name | (define_insn "addqi3" |
| RTL-template | [(set (match_operand:QI 0 "register_operand" "=r,d,r,r") (plus:QI (match_operand:QI 1 "register_operand" "%0,0,0,0") (match_operand:QI 2 "nonmemory_operand" "r,i,P,N"))))] |
| Condition | "" |
| Output | "@ add %0,%2 subi %0,lo8(-(%2)) " |
| | "*return target_output_addqi3(insn,operands,NULL);" |
| Attributes | [(set_attr "length" "1,1,1,1") (set_attr "cc" "set_czn,set_czn,set_zn,set_zn")]) |

**Fig. 10** define_insn for addition arithmetic

Fig. 10 is the addition instruction patterns define_insn including in the standard instruction patterns. The output field in the example is a string that shows how to output matching insns as assembly codes, which are described after a additional notation '@' or what can specify a piece of C codes to compute the output when simple assembly codes substitution can't generate enough expression. For example, you can see the function *target_output_addqi3* in the Fig. 10. '%' in this string specifies where to substitute the value of an operand. Meanwhile, the assembly codes will be produced based on the constraint conditions (constraint in match_operand expression) of the actual operand in operation.

Thought there are many patterns and constraints in the MD, the requirement of the target machine can't be fully met because of the irregular architectures of the actual hardware with different memory configuration and the characteristics of optional peripheral controllers in different embedded SoC. In the Fig. 2, there are many varietas of the memory configuration in different SoCs. For the compiler implementation, we classify

three types as SRSC, MRSC and MRMC off-the-shelf embedded SoCs according to the memory configuration showed in the table 2.

**Table 2**. Classification of memory types.

| Types | Explanation |
|---|---|
| SRSC | Single RAM and Single Code (ROM) |
| MRSC | Multi RAM and Single Code (ROM) |
| MRMC | Multi RAM and Multi Code (ROM) |

This paper uses the C function in the output field explained in Fig 10, to distinguish the different target memory configuration and to decide the correct and optimal generation of assembly codes. In fact, the classification is used in the entire HCC compiler back-end that contains in the target.md, target.h and target.c.

### 5.2 Assembly file structure and machine-dependent processing

HCC compiler translates the source codes to assembly codes using the target.c in compiler back-end to control the assembly file structure. Target.c (*.c) not only provides supports for MD file, but also empowers the further capabilities, which contains the manipulation of assembly file structure, machine-dependent attribute processing such as the processing of new interrupt attribute and so on.

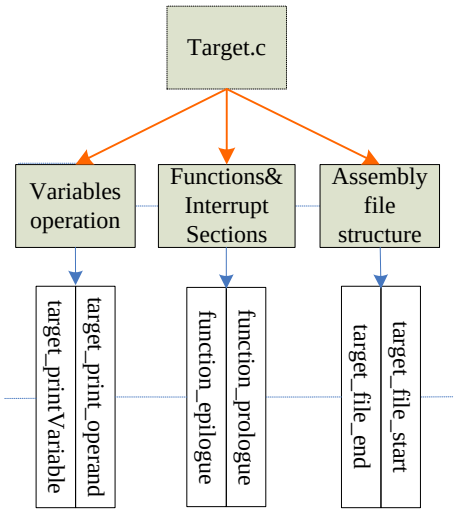
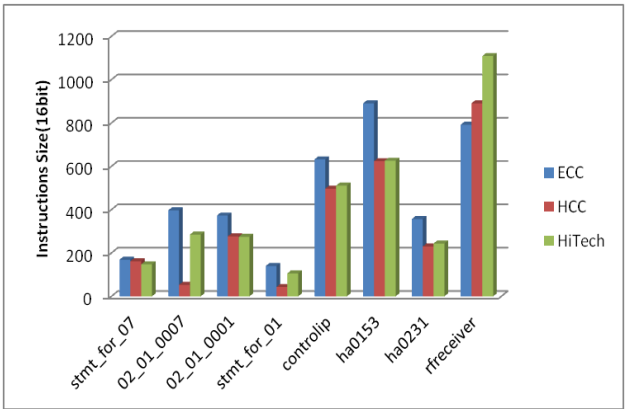



**Fig. 11** Target.c extension and implementation

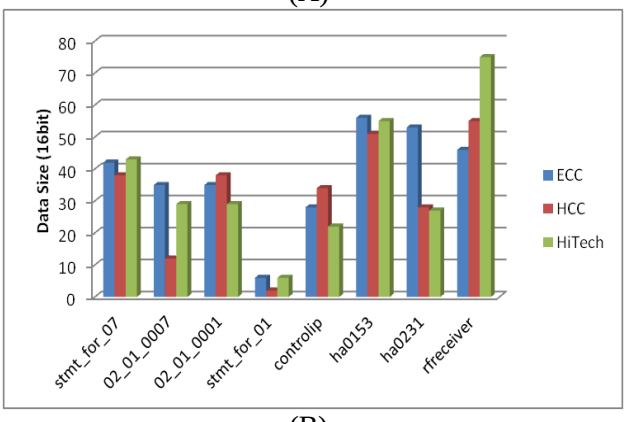
In Fig. 11, the capabilities of target.c are explained as example. HCC uses functions *target_file_start* and *target_file_end* to control the assembly program framework, and use *function_prologue* and *function_epilogue* to implement the section structure for function including saving and recovering environment for interrupt function according to the new interrupt attribute added in Section 4.2.

## 6 Comparisons and Analysis

This paper has chosen two test sets according to the features of the commercial embedded SoCs. One is the GCC TestSuite set which is the standard test suite for GCC compiler. Another one is the commercial programs sets from the practical applications. We complete the standard test and the practical test and compare the test results of HCC, ECC (compiler based on the LCC) and the HiTech complier (HiTech is a complier for the reference embedded SoCs that from professional compiler provider). In the reference SoC structure with multiple banks of program memory and multiple banks of data memory, the performance of HCC exceeds the ECC and the Hi-tech in most of the test programs showed in Fig. 12(A) and Fig. 12(B).




(A)




(B)

**Fig. 12** Assembly codes size of GCC testsuite and customer programs. (A) comparison of the instructions size; (B) comparison of the data size;

The *optimization ratio of codes size* is defined as a metric of assembly codes quality generated from

HCC, ECC and HiTech as shown in Fig. 12. Let, $SI_{HCC}$ = the size of instructions compiled by HCC $SD_{HCC}$ = the size of data compiled by HCC

Use $RI_{HCC\text{-}ECC}$ to express the *instructions ratio* (RI) which is the instructions size comparing of program compiled by HCC and ECC:

$$RI_{HCC\text{-}ECC} = \frac{SI_{ECC} - SI_{HCC}}{SI_{ECC}}$$

If N is the number of test programs, The *average ratio of instruction* (ARI) is thus given by:

$$ARI_{HCC\text{-}ECC} = \frac{1}{N}\sum_{n=1}^{N}\frac{SI_{ECC} - SI_{HCC}}{SI_{ECC}}$$

Corresponding, $ARD_{HCC\text{-}ECC}$ is the *average ratio* of *data* (ARD) which is the data size comparing of programs compiled by HCC and ECC.

$$ARD_{HCC\text{-}ECC} = \frac{1}{N}\sum_{n=1}^{N}\frac{SD_{ECC} - SD_{HCC}}{SD_{ECC}}$$

$AR_{ECC}$ is *average ratio* (AR) of codes size which is the average codes size (instructions and data) comparing of programs compiled by HCC and ECC:

$$AR_{HCC\text{-}ECC} = \frac{1}{N}\sum_{n=1}^{N}\frac{(SI_{ECC} + SD_{ECC}) - (SI_{HCC} + SD_{HCC})}{(SI_{ECC} + SD_{ECC})}$$

$ARI_{HCC\text{-}HiTech}$, $ARD_{HCC\text{-}HiTech}$, $AR_{HCC\text{-}HiTech}$ are the average instructions, data and programs ratio respectively compiled by HCC and HiTech. $ARI_{AVG}$, $ARD_{AVG}$, $AR_{AVG}$ express average numerical results of the instructions, data and programs ratio that HCC compare to ECC and HiTech. Example:

$$AR_{AVG} = \frac{AR_{HCC\text{-}ECC} + AR_{HCC\text{-}HiTech}}{2}$$

**Table 3**. Average optimization ratio of codes size.

| index | ARI | ARD | AR |
|---|---|---|---|
| HCC-ECC | 32.5% | 18.5% | 31.8% |
| HCC-HiTech | 19.8% | 10.1% | 19.1% |
| AVG | 26.2% | 14.3% | 25.7% |

Table 3 shows the caculation results. From the calculation results, we know that HCC produces the smallest assembly codes with the highest assembly codes quality compared with ECC and HiTech. As shown in Table 3, because of the correct implementation and classification for the embedded SoCs and the inherent optimization of GCC, HCC compiler could obtain the average 31% optimization ratio of codes size relative to ECC, average 19% optimization ratio relative to HiTech, and the average 25% codes size decrease compared to compiler ECC and HiTech.

HCC compiler is based on the GCC version 4.2.4 and it can be transplanted from newest GCC version with more advanced optimization. HCC compiler compares to existing compilers for the specific embedded SoCs with more optimization options. Example, ECC is based on LCC compiler which is a lightweight embedded compiler and it hasn't the IR representation (e.g. Gimple, SSA and RTL) with the corresponding optimization.

```
1   #include "HT68F50.h"
2   long int a, b, y;
3   void main(void)
4   {
5         a=0x9a;  //a=0x9a;
6         b=0xa1;  //b=0xa1;
7         b*=a+1;
8         y=a*(a+b);
9         y%=b+2;
10  }
```

**Fig. 13** Compiling effects analysis of example program

Another example is constant folding as shown the program from line 5 to line 9 in the Fig. 13. In this program, line 5 and line 6 have defined the specific values of variables *a*, *b*. HCC could directly use constant folding during the compiling process to get the result of constants multiplication and division, thereby directly calculate the result of variable *y as* the assembly output. But, other compilers such as ECC compiler still need to produce the assembly codes of program statements one by one from line 7 to 9 to calculate the value y with the library supports running in the actual embedded SoCs. So, HCC compared with ECC, and HiTech could omit a lot of program instructions and data during compiling process and it has obvious advantages.

# 7 Conclusion

The HCC complier proposed in this paper not only realizes the language-specific programming syntax of compiler front-end and the machine-dependent back-end design of the complier, but implements the new attribute based on the commercial embedded SoCs, parses and combines the attribute to the corresponding AST syntax tree. This method proposed in this paper could be used for the language-specific programming extension of others' specific embedded SoCs to quickly implement a compiler. For future work, it needs to abstract the more unified extension framework for the specific embedded SoCs with less modifications of GCC. The results have shown that the proposed complier

has achieved excellent performance, which is able to meet the demand of the complier of the specific embedded SoCs. More importantly, the proposed cross complier has been applied to actual products and received a nice feedback from the market.

## Acknowledgements

This research was supported by National Natural Science Foundation of China (General Program) under Grants No. 61274133. The authors would like to thank the engineers from company holtek and all team members' assistance.

*References:*

[1] R. Leupers, M. Hohenauer, J. Ceng, H. Scharwaechter, H. Meyr, G. Ascheid, and G. Braun, Retargetable compilers and architecture exploration for embedded processors, *IEE Computers and Digital Techniques*, Vol. 152, No. 2, 2005, pp. 209-223.

[2] J. Wagner, and R. Leupers, C compiler design for a network processor, *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, Vol. 20, No. 11, 2001, pp. 1302-1308.

[3] N.P. Desai, A Novel Technique for Orchestration of Compiler Optimization Functions Using Branch and Bound Strategy, *Proceedings of IEEE International Conference on Advance Computing Conference (IACC 2009)*, 2009, pp 467-472.

[4] M. Haneda, P.M.W. Knijnenburg, and H.A.G. Wijshoff, Automatic selection of compiler options using non-parametric inferential statistics, *Proceedings of 14th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2005, pp. 123-132.

[5] GNU Compiler Collection. *http://gcc.gnu.org*.

[6] GNU Compiler collection internals, *http://gcc.gnu.org/onlinedocs/gccint*.

[7] X.H.Pei, *Porting GCC to Godson processor*, University of Science and Technology of China, 2010.

[8] K. Ren, X. L. Yan, X. Qin, and L. L. Sun, Design and implementation of a novel ASIP compiler, *Journal of Zhejiang University*, Vol. 42, No. 5, 2008, pp. 553-557.

[9] J.S. Cuadrado, and J.G. Molina, Building Domain-Specific Languages for Model-Driven Development, *IEEE Software*, Vol. 24, No. 5, 2007, pp. 48-55.

[10] G. Hedin, J. Akesson, and T. Ekman, Extending Languages by Leveraging Compilers: From Modelica to Optimica, *IEEE Software*, Vol. 28, No. 3, 2011, pp. 68-74.

[11] L. Antani, H.Ansari, and A.Parameswaran, *Tricore Port for GCC-An Analysis*, Department of Computer Science and Engineering, Indian Institute of Technology, Mumbai, Indian, 2007.

[12] H. Nilsson, *Porting The GNU C Compiler to the CRIS architecture*, Department of Information Technology, Lund Institute of Technology, Sweden, 1998.

[13] R. Trienekens, *Porting the GCC Compiler to a VLIW Vector Processor*, Department of Electrical Engineering, Delft University of Technology, Netherlands, 2009.

[14] G. Fursin, C. Miranda, O. Temam, M. Namolaru, E. Yom-Tov, A. Zaks, and F. Bodin, MILEPOST GCC: machine learning based research compiler, *Proceedings of International Conference on GCC Developers's Summit*, 2008, pp. 7-20.

[15] S.Z. Guyer, and C. Lin, Broadway: A Compiler for Exploiting the Domain-Specific Semantics of Software Libraries, *Proceedings of the IEEE*, Vol. 93, No. 2, 2005, pp. 342-357.

[16] M. Lin, Z.Y. Yu, D, Zhang, Y.M. Zhu, S.Y. Wang, and Y. Dong, Retargeting the Open64 Compiler to PowerPC Processor, *Proceedings of 8th International Conference on Embedded Software and Systems Symposia (ICESS)*, 2008, pp. 152-157.

[17] J.S. Seng, D.M. Tullsen, The effect of compiler optimizations on Pentium 4 power consumption, *Proceedings of 7th International Conference on Interaction Between Compilers and Computer Architectures (INTERA)*, 2003, pp. 51-56.

[18] A. Marongiu, L. Benini, An OpenMP Compiler for Efficient Use of Distributed Scratchpad Memory in MPSoCs, *IEEE Transactions on Computers*, Vol. 61, No. 2, 2012, pp. 222-236.

[19] O. Ozturk, M. Kandemir, G. Chen, Compiler-Directed Energy Reduction Using Dynamic Voltage Scaling and Voltage Islands for Embedded Systems, *IEEE Transactions on Computers*, Vol. 62, No. 2, 2013, pp. 268-278.

[20] K. Kratchanov, T. Golemanov, B. Yksel, and E. Golemanova, Control network programming development environments, *WSEAS Transactions on Computers*, Vol. 13, No. 1, 2014, pp. 645-659.

[21] A. Poggi, Developing scalable applications with actors, *WSEAS Transactions on Computers*, Vol. 13, No. 1, 2014, pp. 660-669.