Measuring the Quality Characteristics of an Assembly Code on Embedded Platforms

Ivan Považan, Miroslav Popović, *Member*, *IEEE*, Miodrag Đukić, and Marko Krnjetin

Abstract **— The paper describes the implementation of a programming tool for measuring the quality characteristics of an assembly code. The aim of this paper is to prove the usability of these metrics for evaluation of an assembly code's quality, which is generated by C Compiler for 32bit DSP architecture. The analysis of test results has shown that the compiler generates a good quality assembly code.**

Keywords **— Assembly code quality, CCC compiler, disassembler, DSP.**

I. Introduction

DIGITAL signal processors - DSP are used as target platforms for implementation of real time digital signal processing algorithms. Fast Fourier Transform, Fast Cosine Transform, filters and codecs are the most common digital signal processing algorithms. Coding of mentioned algorithms represents a hard and expensive job, because it is usually done in an assembly programming language. It is far more favorable and faster to code such algorithms in a higher level programming language. However in that case, high-level compiler needs to be used to transform a high-level language code to a machine code of a target platform.

One such compiler is Cirrus C Compiler – CCC [1], which compiles C source files to assembly code for Cirrus Logic DSP embedded platforms (Coyote 32bit DSP). It has been developed at the Institute for Computer Based Systems RT-RK, Faculty of Technical Sciences in Novi Sad.

The quality of code generated by CCC compiler is examined in this study. Code quality is determined by a comparison of the results of measuring hand-written and compiled codes' quality characteristics.

Quality characteristics are related to hand-written or compiled code's quality referring to the utilization of target platform. Such characteristics are: the frequency of certain instructions usage, the number of memory accesses, the number of conditional and hardware loops, the density of nop instruction (pipeline architectures), the average density of operation per instruction and density of parallel instructions (very large instruction word architectures). Results of measuring such characteristics could refer to the ways of changing the code in order to increase the code's quality. In that manner, usage of these metrics is examined for assessing the assembly code's quality generated by CCC compiler and for improving the compiler in general.

II. Solution Concept

Solution concept is based on two programming tools. In order to understand the case of this study, their description is given below:

- Disassembler – translates a machine code to an assembly language,
- C Compiler – translates C code to a machine code or assembly language.

A. Disassembler

A disassembler is a computer program that translates an executable code to an assembly code. It consists of three components: **Loader**, **Decoder** and **Converter.**

The loader reads an executable file and loads its content to program memory. Program memory represents a memory model of target processor. Content of the input file is being parsed in order to load only useful data to memory. Useful data refers to the machine code of the program which is presented as a list of binary instructions.

The decoder is the main component of the disassembler. It translates an executable code to disassembler's low level intermediate representation (LLIR), based on architecture rules. Instruction set and the architecture are documented by the manufacturer - Cirrus Logic. Decoding consists of analyzing binary instructions in order to determine its instruction type, operation code and used resources. Based on that low level intermediate representation of the analyzed code is being created.

The final phase of translation is emitting the disassembled code to the output file. Emitting is done by the converter whose role is to substitute low level intermediate representation instruction list with appropriate assembly language expressions. Substitution is performed based on instruction type and used resources.

B. C Compiler

A compiler is a computer program that transforms a source code written in a high-level programming language to a lower level language (assembly language or machine code).

The authors received a partial support of the Serbian Ministry of Science and Technology, project number 32031, 2011.

Ivan Považan, Faculty of Technical Sciences Novi Sad, Serbia (e-mail: ivan.povazan@rt-rk.com).

Miroslav Popović, Faculty of Technical Sciences Novi Sad, Serbia (e-mail: miroslav.popovic@rt-rk.com).

Miodrag Đukić, Faculty of Technical Sciences Novi Sad, Serbia (e-mail: miodrag.djukic@rt-rk.com).

Marko Krnjetin, RT-RK Institute for Computer Based Systems LLC Novi Sad, Serbia (e-mail: marko.krnjetin@rt-rk.com).

Compiler consists of two main parts: the front-end and the back-end. Both parts contain a set of programming modules each representing certain compilation phase.

The front-end transforms the initial source code written in a high-level programming language to an abstract syntax tree containing all statements and expressions of the transformed program. The back-end transforms a syntax tree into an intermediate representation (IR), which is not related to any programming language nor to any target platform. The intermediate representation is presented in a form of three-address code [2] (operation list), called high level intermediate representation (HLIR). On the high level intermediate representation, various analyses are being performed, such as: data flow analysis, dependence analysis, function call analysis, function inlining, etc. After analysis phase, follows the instruction selection phase. Instruction selection transforms several elementary operations from high level intermediate representation into an appropriate machine instruction of target processor [3]. Instructions can be presented with parts of operation list of the intermediate representation, called patterns. Therefore, instruction selection depends on choosing the best combination of patterns. The best choice is determined by a desired optimality criterion (e.g.: code size, execution time). The result of instruction selection phase is low level intermediate representation, related to the target platform. The final stages of compilation process are resource allocation, code parallelization, code generation and linking. In the end, an assembly code of the compiled C code is generated.

III. Analyzer

A programming tool for measuring the quality characteristics of an assembly code (hereinafter Analyzer) is based on a disassembler and relies on CCC compiler.

The analyzer consists of several programming modules which represent phases that are related to transformation of a machine code to an assembly language and those that are related to measuring the quality characteristics of a transformed code. The analyzer's job consists of: loading the desired program's machine code, decoding machine instructions, generating disassembler's intermediate representation, transforming disassembler's intermediate representation to compiler's low level intermediate representation and finally measuring the quality characteristics of a given code. Architecture of the Analyzer is shown in Fig. 1. The dashed red line represents the process of translation between different code representations and generation of measurement results.

The transformation of disassembler's intermediate representation to compiler's low level intermediate representation has multiple significance. To start with, it allows the usage of existing compiler's analyses for Analyzer's needs, which simplifies its development. Secondly, it enables future integration between the Analyzer and Cirrus Logic Simulator for Coyote 32bit DSP [4], which could expand Analyzer's functionality. It also enables future integration to CCC compiler, where it could be used for compiler's deficiency detection and its optimization [5]. Finally, transforming disassembler's intermediate representation to compiler's low level intermediate representation enables evaluation of the quality characteristics of an assembly code generated by compiler.

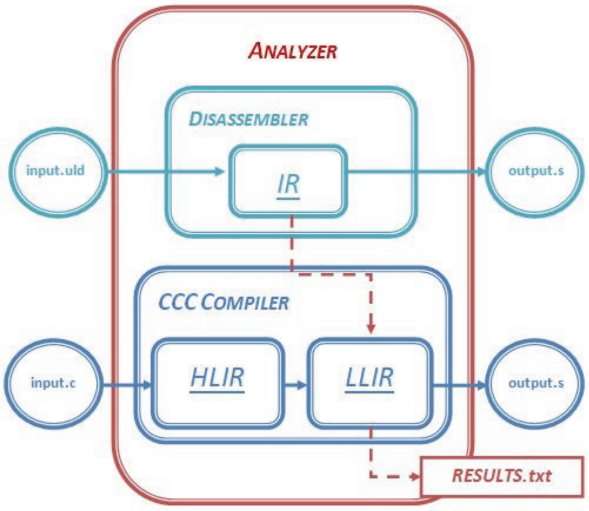


Fig. 1. Flow chart of the Analyzer.

The Analyzer consists of the following programming modules:

- Loader – loads machine code of analysed program,
- Decoder – transforms machine code to disassembler's IR,
- IrConverter – transforms disassembler's IR to compiler's LLIR,
- Measures – measuring quality characteristics of assembly code,
- Main – main programming module.

The following text gives a detailed description of the most important Analyzer's module which refers to measuring the quality characteristics of the assembly code.

A. Measures

Low level intermediate representation, generated from disassembler's intermediate representation, is presented as a list of instructions. It matches the same form of intermediate representation generated during CCC compiler's instruction selection phase. The formed list is used for gathering information that could indicate to quality characteristics of the analyzed code. Codes of interest are algorithm codes written or compiled for Coyote 32bit target processor. Their properties indicate which quality characteristics are relevant. Such characteristics are related to existence of parallelism in the code, usage of hardware loops, usage of certain resources, etc. Therefore counting of certain instruction types, operations and used resources; calculation of desired characteristics and generation of measuring report are being done within the module.

The following code's information is being considered: the number of instructions, the number of parallel instructions, the number of instructions with one, two ormore operations, the number of loops (hardware and conditional), the number of NOP instructions (no operation instructions), the number of indexed memory accesses, the number of index registry loads and the number of function calls and returns. Based on mentioned characteristics, several different calculations are being performed: average density of operation per instruction, density of parallel instructions, percentage of NOP instructions and resource usage.

Code's characteristics data is being gathered by analyzing the instruction list of the intermediate representation. This procedure is being used for all measures except the one related to program loops. The analysis of program loops requires the usage of CCC compiler's routines. The CCC compiler contains a module for program loop analysis which gives information about the number of discovered loops in the analyzed code. To get this information it is only needed for the appropriate routine to be called.

In terms of target processor's utilization, one of the most important code's characteristics is the relation between hardware loops and conditional loops. This is obtained by counting the number of instruction that indicates the beginning of hardware loop and the result of compiler's loop analysis module.

The advantage of using compiler's loop analysis module is providing a simple way of obtaining fundamental information for assembly code's quality analysis. Thereby the significance of using compiler's low level intermediate representation is being revealed. That is the only advantage for now. All analyses supported by the CCC compiler are based on functions in a program. If it could be possible to determine functions in an assembly code, the significance of using compiler's intermediate representation would be even greater.

Finally, when all analyses and measurements have been performed, a report of obtained results is generated as a text file.

IV. Testing

Testing of the Analyzer has been performed with applications' executable files written or compiled for Coyote 32bit DSP target platform in order to evaluate their assembly code's quality. Twenty four tests were performed with algorithms presenting different overlays of DSP firmware and three tests were performed with Transposed Direct Form filter:

- Decoding overlay – 6 algorithms
- Post-processing overlay – 5 algorithms
- Middle-processing overlay – 7 algorithms
- Virtualization overlay – 4 algorithms
- Operating system overlay – 1 algorithm
- Other applications – 1 algorithm
- Transposed Direct Form II – 1 algorithm.

While measuring assembly code's quality of different DSP firmware overlays, 22 hand-written and 2 compiled applications were examined. On the other hand, while testing Transposed Direct Form II filter, 2 hand-written assembly codes and 1 compiled C code were examined.

The results of measuring assembly codes' quality characteristics of different firmware overlay applications are shown in Table 1. Applications of different firmware overlays are colored with different colors. Numbers represent the number of application's test in the same order listed above. The last two tests are related to applications written in C code and compiled with CCC. Due to manufacturer copyrights, the names of tested applications are not specified.

Applications codes' quality is determined by analyzing the obtained results. It is concluded that applications 13, 19 and 20 have the best codes regarding target processor utilization, based on characteristics, such as: density of parallel instructions, percentage of NOP instructions and percentage of hardware loops. Using the same criteria it is concluded that application 22 has the worst code.

The given table shows a huge difference between codes' quality of different algorithms. Therefore the metrics analysis and comparison should be done between the results of the same color (firmware layer). Based on that, it is concluded that among decoders, decoder 5 has the best quality code; among middle-processing applications, application 13; among virtualization applications, application 19 and among post-processing applications, application 8.

By comparison of hand-written and compiled codes it is concluded that among virtualization applications, application 24 has better results than number 18 and comparable results with numbers 19 and 20 regarding percentage of hardware loops. Among middle-processing applications, application 23 has better quality code than number 14 regarding the density of parallel instructions and percentage of NOP instructions, while compared to other applications of the same overlay it has a slightly worse code.

For measuring assembly code's quality of applications written by C programming language, it was expected that it would give poor results. The complexity of compilation process and generating the optimized code regarding target processor utilization represent a serious challenge for compiler development [6]. Despite those problems, by analyzing test results, it may be concluded that CCC compiler generates a good quality assembly code. A good quality code is the one with a high density of parallel instructions and a high percentage of hardware loops on the one hand, and as few as possible NOP instructions on the other. Such a code is considered optimized for execution on the target platform.

Results related to the numbers of function calls and returns show a huge difference between obtained values. Application numbered 6 has a more than three times higher number of function calls than returns from functions. This indicates the difficulty of function detection in an assembly code. Several reasons for this are: the existence of index registry function call, the existence of multiple returns from a function or existence of different overlay function call.

Besides testing the quality of different firmware overlay applications' code, testing of hand-written and compiled code of the same algorithm has been performed. For these additional test cases, the algorithm of Transposed Direct Form II filter has been chosen. Two hand-written codes and one compiled code have been analyzed. C code of the algorithm is given below, as well as measuring results (Fig. 2, Table 2).

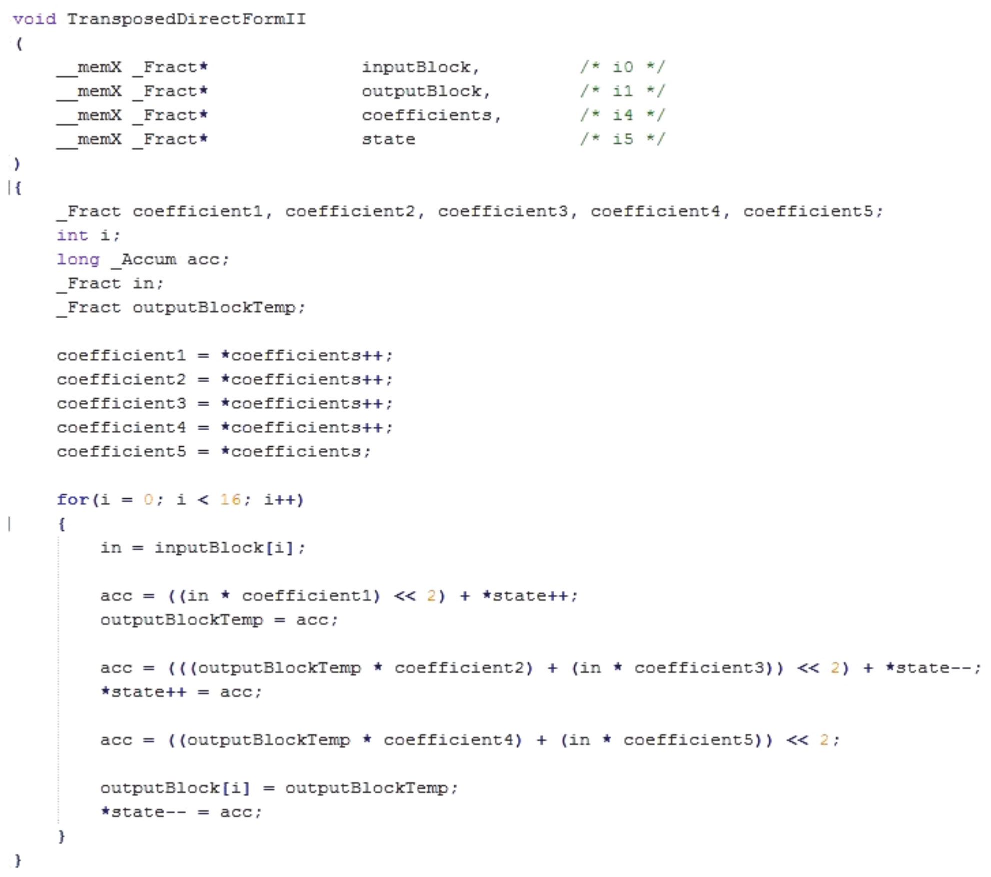
Hand-written codes were written by senior software engineers, who were given an hour to write and optimize the assembly code. On the other hand, the shown C code was compiled with CCC compiler in five seconds. The importance of this test is comparison of quality between hand-written and compiled code and time spent for their writing and compiling. By analyzing the results it may be concluded that using CCC compiler is far more favorable.

TABLE 1: DSP APPLICATIONS TEST RESULTS.

#	*Instr. count*	*Parallel instr. density*	*NOP instr. percentag*	*Index registry loads*	*Indexed-memory access*	*Functio n call count*	*Function return count*	*Loop count*	*Hardware loop percentage*
1	4384	11.59%	2.21%	450	472	249	68	514	19.46%
2	4952	7.84%	1.92%	488	399	296	101	533	20.64%
3	6025	12.66%	1.96%	859	731	420	146	768	29.65%
4	16954	13.47%	0.96%	1830	1977	950	337	2638	94.96%
5	6427	16.56%	0.23%	371	622	297	121	947	96.61%
6	12683	9.03%	1.19%	1119	1215	796	255	1851	26.04%
7	3913	12.96%	0.69%	184	534	135	73	226	20.96%
8	3614	17.13%	0.80%	413	565	150	111	139	20.95%
9	8221	13.90%	1.09%	846	1043	261	179	800	26.75%
10	5342	16.47%	0.77%	673	856	226	166	706	29.89%
11	6232	15.76%	0.71%	761	963	240	177	844	28.32%
12	3908	18.47%	0.56%	370	701	96	70	326	100.00%
13	2678	19.94%	0.37%	336	433	104	71	111	98.21%
14	12141	11.37%	1.93%	1230	1272	709	441	1315	20.45%
15	3646	15.63%	0.85%	244	608	114	101	489	13.91%
16	3903	15.48%	0.87%	283	646	126	108	389	20.31%
17	1216	13.16%	0.41%	128	171	24	22	42	20.00%
18	1775	14.99%	1.63%	273	290	107	65	63	60.71%
19	663	23.23%	0.30%	84	156	26	21	36	95.24%
20	1273	21.92%	0.00%	209	295	48	22	56	78.69%
21	12824	10.32%	1.78%	1032	1174	812	485	1423	53.58%
22	765	3.40%	1.44%	57	50	155	56	70	13.54%
23	8727	13.53%	1.05%	674	1350	361	163	543	43.46%
24	2886	15.38%	0.87%	301	501	72	48	150	80.67%

TABLE 2: TRANSPOSED DIRECT FORM II TEST RESULTS.

Transposed Direct Form II							
Test	*Instr. Count*	*Parallel instr. count*	*Parallel instr. density*	*Index registry loads*	*Indexed memory access count*	*NOP count*	*Hardware loop count*
Hand-written 1	39	11	28.2%	0	11	0	1
Hand-written 2	47	13	27.66%	4	12	0	1
Compiled	38	14	36.84%	0	11	0	1



```
void TransposedDirectFormII
(
    __memX _Fract*              inputBlock,         /* i0 */
    __memX _Fract*              outputBlock,        /* i1 */
    __memX _Fract*              coefficients,       /* i4 */
    __memX _Fract*              state               /* i5 */
)
{
    _Fract coefficient1, coefficient2, coefficient3, coefficient4, coefficient5;
    int i;
    long _Accum acc;
    _Fract in;
    _Fract outputBlockTemp;

    coefficient1 = *coefficients++;
    coefficient2 = *coefficients++;
    coefficient3 = *coefficients++;
    coefficient4 = *coefficients++;
    coefficient5 = *coefficients;

    for(i = 0; i < 16; i++)
    {
        in = inputBlock[i];

        acc = ((in * coefficient1) << 2) + *state++;
        outputBlockTemp = acc;

        acc = (((outputBlockTemp * coefficient2) + (in * coefficient3)) << 2) + *state--;
        *state++ = acc;

        acc = ((outputBlockTemp * coefficient4) + (in * coefficient5)) << 2;

        outputBlock[i] = outputBlockTemp;
        *state-- = acc;
    }
}
```

Fig. 2. Transposed Direct Form II C cod.

V. Conclusion

Using subjective verification comparing the obtained results and the assembly code of analyzed application it is determined that the system works correctly. After analyzing test results it is concluded that CCC compiler generates a good quality assembly code.

A basis for future Analyzer's improvement is transforming the code to low level intermediate representation of the compiler. With that, several ways of expanding the tool are possible:

- Integration of the tool to CCC compiler in order to expand compilers functionality. Integration would require changes related to transforming the machine code directly to a compiler's low level intermediate representation
- Implementation of an inverse transforming process which would form a high level intermediate representation based on a low level one. This would open a possibility of transforming a machine code of one processor to another with a different architecture – (cross-assembling).
- Integration between the tool and the Cirrus Logic Simulator for Coyote 32bit DSP [4], to enable a dynamic program analysis. This would open a possibility of detecting functions in an assembly code. Analyzing code's quality of each function regarding resource usage could indicate a better solution of resource allocation. In that case a programming tool for measuring the assembly code's quality would participate in generating an optimized compiled code.

References

[1] Z. Zarić, M. Đukić, M. Gajić, and M. Popović: *Jedan koncept optimizacione tehnike za iskorišćenje adresne jedinice u C kompajleru,* Fakultet tehničkih nauka, Odsek za računarsku tehniku i računarske komunikacije, Novi Sad, 2008.

[2] Aho, A.V., R. Sethi, and J. D. Ullman: *Compiler: Principles Techniques, and Tools,* Addison-Wesley, Reading MA, 1986.

[3] V. Kovačević and M. Popović: *Sistemska programska podrška u realnom vremenu*, Univerzitet u Novom Sadu, Fakultet Tehničkih Nauka, 2002.

[4] M. Đukić, N. Četić, R. Obradović, and M. Popović: *An Approach to Instruction Set Compiled Simulator Development Based on a Target Processor C Compiler Back-End DesignI,* Faculty of Technical Sciences, Novi Sad, 2009.

[5] J. Bungo: *The Use of Compiler Optimizations for Embedded Systems Software,* USA, 2008.

[6] Fisher, J. A., P. Faraboschi, and C. Young: *Embedded Computing: A Vliw Approach To Architecture, Compilers And Tools,* Morgan Kaufmann, San Francisco CA, USA, 2005.