http://ansinet.com/itj

ITJ

ISSN 1812-5638

INFORMATION TECHNOLOGY JOURNAL

ANSInet
Asian Network for Scientific Information
308 Lasani Town, Sargodha Road, Faisalabad - Pakistan

Information Technology Journal 13 (13): 2145-2151, 2014
ISSN 1812-5638 / DOI: 10.3923/itj.2014.2145.2151
© 2014 Asian Network for Scientific Information

Build and Optimization Method of Embedded Cross Assembler

[1,2]Liu Gao-ming, [2]Li Chao, [2]Liu Song-chao and [2]Tan Huai-liang
[1]Changsha Vocational Technical College, Changsha, 410217, China
[2]College of Information Science and Engineering,
Hunan University, Yue Lu Shan Road, Changsha, 410082, China

Abstract: Since embedded technology have been proposed and quickly used in many fields, cross-assembler become a hot method in whole industry. The method for building rapidly assembler was proposed to solve the traditional assembler defects, such as long development cycle, high development costs, being incompatible with the new types of microcontrollers and so on. Also, the traditional uses fixed-size sliding window which makes it hard to match and optimize instructions sequence. An improved peephole optimization method which uses unfixed sliding window and can easily identify the non-continuous sequence of instructions, was designed by using regular expression. The experimental results showed that it can efficiently improve the assembler's space-time performance. Therefore, the approaches proposed in this study optimize the performance of assembler obviously.

Key words: Assembler build, optimization, unfixed sliding window, dynamic configuration

INTRODUCTION

In the course of designing of embedded products, cross-assembler plays an important role. How to develop cross-assembler fast and efficiently is essential to the whole industry. Also, due to the strict requirements of embedded products, the research on how to improve the cross-assembler's optimization ability is becoming more and more popular. Now-a-days, there are some very new and popular optimization techniques, such as optimization sequences searching, optimization based on machine learning and parallel optimization (Chabbi *et al*., 2011; Agakov *et al*., 2006; Joisha *et al*., 2011), etc. Peephole optimization, a kind of optimization technique, plays an important role in improving applications' performance. However, in traditional peephole optimization, the size of the sliding window is fixed or has upper limit which doesn't work well in identifying a non-continuous sequence of instructions. So, it is necessary to improve the traditional peephole optimization techniques. In this study, we described how to develop cross-assembler in a relatively fast speed and presented a new method for peephole optimization.

There are two ways of developing assembler. One is developing every part of it with the help of a variety of tools and the other way is porting. Flex and bison (Kesar, 2008) are two famous development tools and are used to generate lexical analyzer and parser, respectively. Aaby (2004) described how to develop compiler by Flex and Bison. Chen *et al*. (2006) also gave a method for developing assembler rapidly with Flex and Bison. These approaches all require a clear understanding of the rules and principles of Flex and Bison. However, considering that some languages' lexical and syntax rules are not easy to describe, therefore, it is ineffective to use these approaches sometimes. In contrast, software porting has many advantages. It not only reduces the workload significantly, but also maintains almost all the functions of the original assembler. What's more, the chance of error is reduced because the original assembler is repeatedly tested. Li (2004) described how to port GNU AS and GNU LD to the MIPSX-based platform.

Based on the above analysis, we developed the cross-assembler whose instruction set can be dynamically configured and extended by improving GNU open source software GPASM (Gputils, 2011). The peephole optimization method, whose sliding window can be extended and can easily identify the non-continuous sequence of instructions, was proposed by taking full advantage of regular expression. The two schemes can efficiently improve the assembler's space-time performance. With the assembler, we will show the advantage of this study proposed approaches which improve the system performance.

Corresponding Author: Liu Gao-ming, Changsha Vocational Technical College, Changsha, 410217, China

GPASM FRAMEWORK INTRODUCTION

Base data structure of GPASM:

- **GPASM_state:** It records almost all the information the assembler needs at compile time, such as parameters of the command line, initialization information, various symbol tables, pointers to the input and output files, etc. This way of storage is helpful to data search and unified management
- **Macro struct:** It is divided into macro head and body and stored in macro_head and macro_body, respectively. The macro_head using generalized table to store the macro parameters and singly linked list structure is used by macro_body that one node to store one line statements
- **Memblock:** It is used to store and manage the generated machine code and finally output them to the target file. Memory blocks organization is used by memblock, the size of each block is 32K and each memory block is managed by a corresponding memblock structure

Structure and working process of GPASM: The structure of GPASM is depicted in Fig. 1.

The working process of GPASM is described as following:

- **Initialization:** The initialization begins before compilation. It reads environment variables and does basic information configuration, such as compilation mode is absolute addresses or relocatable, whether the case is sensitive or not, etc. All these information is recorded in the corresponding members of GPASM_state
- **Reading the source file:** The command line is processed by function process_args (int argc, char *argv[]). It reads various parameters from the command line (Including the file path), analyses them and records the results in corresponding members of GPASM_state
- **Compilation:** The compilation is realized by function assemble (void). During the first pass, all the labels are marked and recorded in various symbol tables. When it comes to the second pass, lexical analysis and syntax analysis start to work. If both of these processes are well done, the parser calls corresponding functions to do semantic analysis. After that, the machine code will be generated and stored in memory blocks managed by data structure MemBlock. At last, after all these processes, the machine code will be written into different kinds of files (.cod, .lst, .hex)

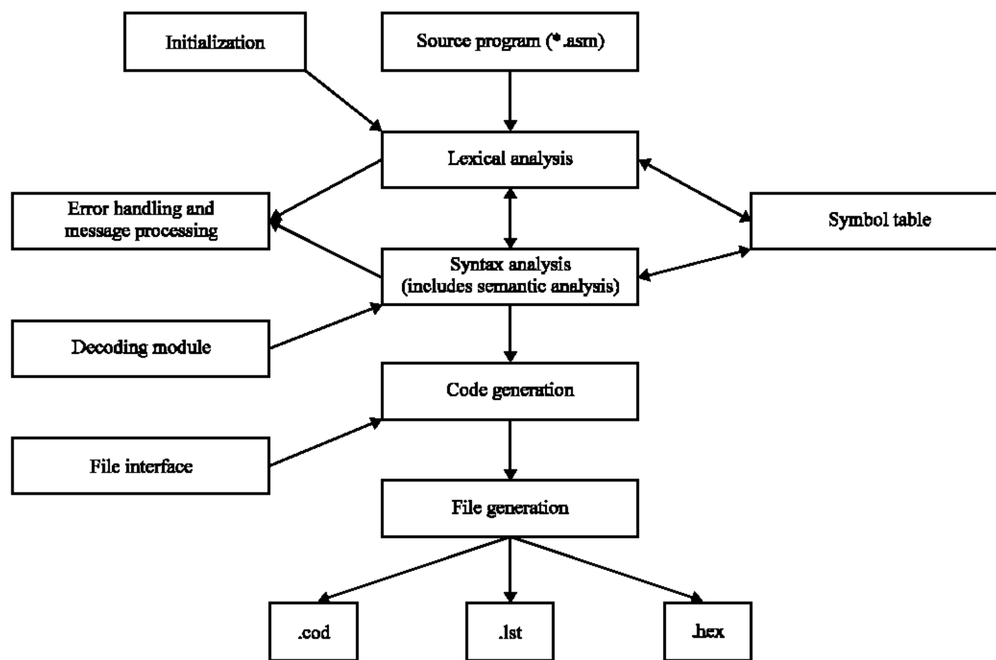

Fig. 1: Structure of GPASM

BUILD METHOD OF CROSS ASSEMBLER

The cross assembler which can be dynamically configured and extended consists of the following four parts.

Generation of lexical analyzer: There are two common ways to generate lexical analyzer. One way is to write the configuration file (*.l) according to the requirement of new language and then the lexical analyzer is generate by using flex. The other way is to modify the original configuration file (*.l) to make it suitable for the new language's trait. Considering the efficiency and compatibility in practice, we chose the latter one. For the morpheme that is not contained by new language, delete it; and for the new morpheme, add it into configuration file legally. Because lexical analysis and syntax analysis are closely linked to each other, some morphemes' information is recorded in parser. This makes it important to consider the influence to parser caused by the change of some morphemes and modify the syntax configuration file if necessary. For example:

```
List
{
yylval.s = strdup(yytext);
return LIST;
}
```

The above code is an interception from the lexical configuration file (*.l). In this case, if keyword list is not contained in the new language, it should be deleted from the configuration file. But it's not enough. List is also contained in the syntax configuration file (*.y): #define LIST 265. So, it also should be deleted from here. Number 265 can be retained for other new morpheme.

Generation of parser: The parser was generated by taking the same strategy as the generation of lexical analyzer, that is, by modifying the configuration file (*.y) to make it support the new language. The priority of the operators needs to be considered when modifying the configuration file (*.y). The higher priority the operator has, the lower it is put in the rule part of the file. Considering the requirements of the project, some new operators are added to the new assembler in the process of generating it, such as "+=", "-=", etc. The grammar rules need to be modified to support the new syntax elements. According to the regulation, "+=" should be on top of "+" because of the priority of operator "+=" is lower than operator "+". As shown follow:

```
e4:   e3 |
      e4 e3op e3
      {
coerce_str1($1); coerce_str1($3);
$$ = mk_2op($2, $1, $3);
      };
e4op: '+=' | '-=' ;
e3:   e2 |
      e2 e3op e2
      {
coerce_str1($1); coerce_str1($3);
$$ = mk_2op($2, $1, $3);
      };
e3op: '+' |'-';
```

The above code is an interception from the syntax configuration file (*.y). ei(i = 1, 2, 3…) stands for expression and eiop(i = 1, 2, 3…) stands for operator. These simple arithmetic expressions will later be optimized by live variable analysis and available expressions analysis.

More other operators can be added with similar method. Generated syntax tree after syntax analysis is consist of binary tree. In order to build a new assembler effectively and low cost, some corresponding necessary modifications should be made and the original processing function such as the syntax tree generation function, macro structure processing function, machine code generation function, etc should be used as far as possible.

Code generation: Generation of object code is mainly decided by file directive.c. Most functions in this file are named by pattern "do_name of instruction", such as do_equ, do_db, do_set, etc., as backend interface function of specific target machine. These functions are called by parser to do semantic analysis and generate machine code. To every instruction, its corresponding name, machine code (in hexadecimal form) and type are stored in an array:

```
struct insn op_16cxx[] = {
{"addlw", 0x3e00, 0x3e00, INSN_CLASS_LIT8 },
{"addwf", 0x3f00, 0x0700, INSN_CLASS_OPWF7 },
{"andlw", 0x3f00, 0x3900, INSN_CLASS_LIT8 },
{"andwf", 0x3f00, 0x0500, INSN_CLASS_OPWF7},
{"bcf", 0x3c00, 0x1000,INSN_CLASS_B7},
…
}
```

The code listed above is a part of the 14 bit instructions array. There are also other kinds of arrays to record different types of instructions. In the study's assembler, the instructions' information was added to the corresponding array to support the new type of microcontroller.

Dynamic binding of instruction set: Sometimes different types of microcontrollers have different instructions and corresponding machine code. In order to support different kinds of microcontrollers and the new series not available at present, the parts closely linked to the hardware should be detached from the assembler. This makes the assembler more general.

From the working process of GPASM, it is not difficult to find that the instruction set part is loosely coupled with other parts of the program. So, the instruction set part can be detached from the assembler and stored in a separate file. Considering that during the compile time only one type of microcontrollers will be used, the instruction sets are stored in separate files named by their types. To keep the instruction sets safe and unchangeable, the instruction set part was stored in DLL files. When the assembler works, it reads the instruction set from the corresponding file and dynamically binds it.

OPTIMIZATION METHOD

There are high-level and low-level optimizations. The high-level optimization usually is machine-independent. It includes common sub-expression elimination, copy propagation, strength reduction of induction variables, loop optimization, etc. However, the low-level optimization is hardware related. It includes register allocation, instruction scheduling, etc. Both the high-level and low-level optimizations depend on the support from data flow analysis which is the basis for a variety of compiler optimizations.

Data flow analysis collects data's information along the possible execution paths of the program. It analyses variable's definition and usage through algebraic method and collects semantic information from the code for better optimization (Fang *et al*., 2009). There are many solutions to the classic and recent problems in the data flow analysis field (Jayaseelan *et al*., 2010; Fosdick and Osterweil, 1976; Muchnick, 1997; Randy *et al*., 1986; Stevens and Jones, 1981; Allen and Kennedy, 2001) and the mainly used method is to establish data flow equations. In this assembler, live-variable analysis and available expressions analysis were applied to help optimizing programs.

Live variable analysis: Given a point p and variable x, it is wished to know whether the value of x at p could be used along some path in the flow graph starting at p. If so, it demonstrates x is live at p; otherwise, x is dead at p (Aho *et al*., 2007). A reference to a variable makes it live while a definition of a variable makes it dead. Live variable analysis is a backwards data flow analysis and usually used in register allocation for basic blocks. If a variable is dead at the end of a basic block, then there is no need to store the variable in register and the definition of the variable can be deleted. For a given basic block A, some sets are defined as follows:

In[A]:	The set of variables live at the points immediately before A
Out[A]:	The set of variables live at the points immediately after A
Def[A]:	The set of variables defined prior to be referenced in A
Use[A]:	The set of variables referenced prior to be defined in A

The corresponding data flow equations are:

$$In[A] = Use[A] \cup (Out[A]-Def[A])$$
$$Out[A] = \cup\, In[P] (P \text{ is A's successor})$$

The first equation means that, if a variable is live at the point immediately before a basic block, it is either referenced prior to its definition or it is live at the point immediately after the block and is not defined in the block. The second equation means that, a variable is live at the point immediately after a basic block if and only if it is live in one of the block's successors.

Algorithm: Live-variable analysis

Input: The control flow graph the program with Def and use sets computed for every block

Output: In and Out sets for each basic block

Method: In[EXIT] = ∅;/*ENTRY and EXIT represents the beginning and end of the flow graph, respectively*/

```
for (every basic block A other than EXIT) In(A) = ∅
while(changes to any block's In set)
{
    for (every basic block A other than EXIT)
    {
    Out[A] = ∪ In[P] (P is A's successor)
    In[A] = Use[A] ∪ (Out[A]-Def[A])
    }
}
```

Available expressions analysis: An expression x op y is available at point p if every path from the entry node to p evaluates it and doesn't change x or y from the last evaluation point of the expression to p (Xiaofei, 2007). If x or y is changed in a block and the expression x op y doesn't be recomputed, this demonstrates say the block kills the expression.

The available expressions analysis is mainly used to detecting global common sub-expressions. If an expression is available, the result is kept to replace the same expression when it is encountered next time instead of computing it once again. Suppose E is the set of all the expressions appearing on the right of the statements of the program. For a basic block A, some sets are defined as follows.

In[A]: The set of available expressions live at the points immediately before A
Out[A]: The set of available expressions live at the points immediately after A
Gen[A]: The set of expressions generated by block A
Kill[A]: The set of expressions in E killed by block A

The corresponding equations are:

Out[A] = Gen[A] ∪ (In[A]-Kill[A])
In[A] = ∩ Out[m] (m is A's predecessor)

The meaning of the equations is easy to understand.

Algorithm: Available expressions analysis

Input: A flow graph with Gen and Kill sets computed for every block
Output: In and Out sets for every block
Method:

```
Out[ENTRY] = ∅
for (every basic block A other than ENTRY) Out(A) = E
while(changes to any block's Out set)
{
      for (every basic block A other than ENTRY)
      {
In[A] = ∩ Out[m] (m is A's predecessor)
      Out[A] = Gen[A] ∪ (In[A]-Kill[A])
      }
}
```

Peephole optimization: The traditional peephole optimization uses template to match different sequences of instructions (Gang, 2004). Such as the compiler GCC, when it scans and optimizes code, if the instructions in the sliding window meet the form defined in the template, they will be replaced by a shorter or faster sequence. However, it doesn't work well in matching and optimizing a non-continuous sequence of instructions. In this study, a simple and efficient way was proposed. It was implemented by using regular expression and Flex and Bison can either be a separate part or embedded in the lexical analyzer. The detail descriptions of this method and some peephole optimization strategies designed in this new assembler for optimizations are as follows:

- Instruction "movff A, B" means directly transferring data from file register A to file register B. This instruction is not included in the early low-level PIC microcontroller series. In these early series, working register w is occupied every time when data from file register A needs to be transferred to file register B. Also, it have to ensure that the data in w is useless; otherwise it should be saved in a temporary file register first. For example:
  - movwf temp
  - movf port c, w
  - movwf port b
  - movf temp, w

The above code means transferring data from port c to port b with the help of working register w. Before the transmission, working register w needs to be saved in a temporary register first for later use. It seems bulky that the data from port c should be transferred to working register w first, then from w to port b. So, when the programs of the early microcontroller series run on the later series that supporting instruction movff, the above instructions can all be replaced by just one: Movff port c, port b. This way obviously saves time and space.

In this assembler, regular expression is used to match the above instructions. The corresponding regular expression is:

```
[\t]*"movwf"[\t]*.*[\n][\t]*"movf".*[,][\t]*w.*[\n][\t]*"
    movwf"[\t]*.*[\n][\t]*"movf".*[,][\t]*w.*[\n]
```

[\t]* means all the spaces and tabs,.* means all the characters except the new line character and [\n] means the newline character.

Then the regular expression was written into the configure file (*.l) to automatically generate program by Flex. The program was embedded into the assembler and started to work during the optimization process. When the above instructions were matched, the content from pointer yytext was analyzed and replaced by a new simple string.

Before the peephole optimization, the assembler only did lexical and syntax analysis. When the peephole optimization was over, the assembler did lexical and syntax analysis once again. This time, the assembler went on to do semantic analysis and data flow analysis to further optimize the program through live variable analysis and available expressions analysis. Finally, after all these processes, the object code generation began. To support the above sequence of processes, the structure of the GPASM needs to be changed. One more lexical and syntax analysis were added.

- The above instructions show that the sliding window's size is four. However, sometimes the instructions are not put together. For example:
  - movf port c, w
  - nop
  - …
  - movwf port b

In such case, it is difficult to figure out the size of the sliding window and match the instructions. Before further discussion, it can be assume that working register w doesn't appear between instructions movf port c, w and movwf port b; otherwise it is of no meaning. To match

such a non-continuous sequence of instructions, the configuration file(*.l) of Flex should be modified as follows:

- declaration part
- %%
- translation rules part
- %%
- auxiliary functions part

First, add to the declaration part definition: Oneline [^w]*\n. oneline means a line without character w.

Then add to the translation rules part regular expression and corresponding actions. The regular expression is:

```
[\t]*"movwf"[\t]*.*[\n]{oneline}*[\t]*
"movf".*[,][\t]*w.*[\n]{oneline}*[\t]*"movwf"[\t]*.*
[\n]{oneline}*[\t]*"movf".*[,][\t]*w.*[\n]
```

The regular expression means inserting zero or more lines without character w between the instructions.

- In some programs, its often see the "write immediately after read" operation. For example:
  - movf port c, w
  - movwf port b

The above instructions mean reading the data from port c and writing it to port b immediately. However, as a certain delay sometimes occurs between circuits, the above instructions may not run well. A solution is to insert a "nop" instruction between the instructions to give a little pause. So, the above instructions can be transformed to:

- movf port c, w
- nop
- movwf port b

Flex is also used to optimize the above code. The corresponding regular expression is: "movf".*[,][\t]*w[\n][\t]*"movwf"[\t]*.*[\n]. The optimization is controlled by a command-line parameter. This strategy fits the low-level microcontrollers that don't run fast enough.

EXPERIMENTAL RESULTS

To measure the performance of the optimization, two sets of tests have been ran. The platform consists of Intel dual-core microprocessor T1080 1.73 Ghz, 1G memory and windows XP. The object code ran on simulator GPSIM. Before each simulation, all other applications were closed to rule out the influences from these applications.

Table 1: Size comparison of different programs

Source code	GNU GPASM	Study's compiler Un-optimized	Study's compiler Optimized
Program 1	867	865	829
Program 2	495	491	491
Program 3	1.41k	1.39k	1.35k
Program 4	1.97k	1.95k	1.92k
Program 5	2.35k	2.32k	2.32k
Program 6	3.19k	3.15k	3.04k
Program 7	1.20k	1.18k	1.14k
Program 8	1.96k	1.92k	1.83k

Unit: byte

Table 2: Runtime of different programs

Source code	GNU GPASM	Study's Compiler Unoptimized	Study's Compiler Optimized
Program 1	7.4	7.2	6.8
Program 2	20.6	20.4	20.4
Program 3	21.3	20.8	20.4
Program 4	27.8	27.2	26.8
Program 5	41.9	41.4	41.4
Program 6	51.7	50.8	49.6
Program 7	27.7	27.2	26.8
Program 8	36.1	35.6	34.6

Unit: μsec

The first set of the tests compared the size of the program optimized by using the study's strategies with of those unoptimized ones. The tested programs are common ones and most of them are in frequent use. Table 1 shows the result of the test.

Table 1 shows that the study's un-optimized compiler can effectively reduce objective file size about 1.1-2.4%, especially the optimized compiler can bring 7% size decrease comparing with GNU GPASM (Gputils, 2011) for some applications, such as program 1, program 3, program 6 and program 8. The optimization is effective and the size was reduced by 1.5-4.6%. It is because in these applications, there are direct data transfer operations from register to register and the operations meet the study's optimization strategies. It believe that in those programs that have more data transfer operations, the optimization will be more effective. For other applications, such as program 2 and program 5, the optimization is ineffective. It found that in these applications, there are no direct data transfer operations from register to register. So, the study's optimizations could do nothing.

The second set of the tests compared the runtime of the programs. The results came from simulator GPSIM. GPSIM is a piece of open source software and used to simulate programs run on PIC based architectures. After every simulation, the simulator rebooted to rule out all the other influences. Table 2 shows the runtime of the programs.

The following Table 2 shows that the study's optimized compiler can distinctly lower runtime by 8%, the un-optimized compiler also can bring 3% runtime decrease comparing with GNU GPASM (Gputils, 2011) for some applications, such as program 1, 3, 6 and 8. The optimization is effective and the runtime was reduced by 1.5-5.6% in program 1, 3, 4, 6, 7 and 8. It seems interesting that the runtime of program 2 is much longer than that of program 1 but program 2 has smaller size. That's because program 2 is mainly consists of loops while program 1 doesn't.

CONCLUSION

In this study, a building method of cross assembler and a approach for peephole optimization are presented. This study uses unfixed sliding window and can easily identify the non-continuous sequence of instructions. And the study developed the cross-assembler whose instruction set can be dynamically configured and extended by improving GNU open source software GPASM. The experimental results demonstrate that to some extent these strategies are useful and effective. To further the research on optimization, our next emphasis will focus on how to apply those to new techniques, such as iterative optimization, self-adaptive optimization and optimization based on machine learning, to the traditional compiler design.

REFERENCES

Aaby, A.A., 2004. Compiler construction using flex and bison. http://foja.dcs.fmph.uniba.sk/kompilatory/docs/compiler.pdf

Agakov, F., E. Bonilla and J. Cavazos, 2006. Using machine learning to focus iterative optimization. Proceedings of the International Symposium on Code Generation and Optimization, March 26-29, 2006, New York, USA., pp: 295-305.

Aho, A.V., M.S. Lam, R. Sethi and J.D. Ullman, 2007. Compilers: Principles, Techniques and Tools. 2nd Edn., Addison-Wesley, New York, USA.

Allen, R. and K. Kennedy, 2001. Optimizing Compilers for Modern Architectures. Morgan Kaufmann Press, San Francisco, CA., USA.

Chabbi, M.M., J.M. Mellor-Crummey and K.D. Cooper, 2011. Efficiently exploring compiler optimization sequences with pairwise pruning. Proceedings of the 1st International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era, June 5, 2011, San Jose, California, USA., pp: 34-45.

Chen, H.B., C.L. Liu and D.L. Hu, 2006. A method for quickly constructing an assembler and its application. Comput. Eng. Sci., 6: 131-134.

Fang, X., Q. Yin, L.H. Jiang, H. Huang and H.Q. He, 2009. Register parameter recovery method based on data flow analysis. Comput. Eng., 22: 62-64.

Fosdick, L.D. and L.J. Osterweil, 1976. Data flow analysis in software reliability. ACM Comput. Surveys, 8: 305-330.

Gang, F., 2004. Research and development of the cross compiler based on GCC for embedded system. M.Sc.Thesis, Zhejiang University, China.

Gputils, 2011. Gnu PIC utilities. http://gputils.sourceforge.net/

Jayaseelan, R., A. Bhowmik and R.D. Ju, 2010. Investigating the impact of code generation on performance characteristics of integer programs. Proceedings of the Workshop on Interaction between Compilers and Computer Architecture, March 13-17, 2010, Pittsburgh, PA., USA.

Joisha, P.G., R.S. Schreiber and P. Banerjee, 2011. A technique for the effective and automatic reuse of classical compiler optimizations on multithreaded code. Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, January 26-28, 2011, Austin, TX, USA., pp: 26-28.

Kesar, S.K.S., 2008. Semi-automated software restructuring. M.Sc. Thesis, Syracuse University, New York, USA.

Li, X.B., 2004. The porting of cross-assembler and cross-linker based on GNU binutils. Ph.D. Thesis, Zhejiang University, China.

Muchnick, S., 1997. Advanced Compiler Design and Implementation. Morgan Kaufmann Publishers, Burlington, MA., USA., ISBN-13: 9781558603202, Pages: 856.

Randy, A.J., D. Callahan and K. Kennedy, 1986. An implementation of interprocedural data flow analysis in a vectorizing fortran compiler. Technical Reports, Department of Computer Science, Rice University.

Stevens, M. and N.D. Jones, 1981. Program flow analysis: Theory and applications. Prentice-Hall, Englewood Cliffs, NJ., USA.

Xiaofei, W., 2007. Research and cases study on data flow analysis. Ph.D. Thesis, National University of Defense Technology, China.