A Study on GCC Optimizer

Lei Ding, Jie Yao
Computer Science Department, University of Arizona

Abstract

The goal of GCC is to produce faster and compact code. It provides five levels of optimization options. In order to evaluate differences among these options, we use a scientific way to study it. Firstly, we proposed our hypotheses based on the common sense. Then we designed our experimental plan and developed the infrastructure. After that, we performed the experiment and analyzed the results. Additionally, we took efforts to modify the original hypotheses based on our findings. Finally, we concluded the limitations of our study which are worthy of further study.

The most important thing we learned from this study is how to do research. What's more, we know that experiment plays really important role in research. As PhD candidates, we sincerely believe that we benefit a lot from this scientific approach.

1. Introduction

The GNU project was started in 1984 to create a complete Unix-like operating system as free software, in order to promote freedom and cooperation among computer and programmers. The first release of GCC is made in 1987. This was a significant breakthrough in C compiler, being the first portable ANSI C optimizing compiler which is released as free software. Since then GCC has become one of the most important tools and most widely used C compiler in the development of free software. GCC is an optimizing compiler. It provides a wide range of options for difference purposes, like increasing the speed, reducing the size of the executable files it generate. In this paper, we treat the GCC as a black box and study its optimizer in a scientific way. Our basic idea is to compare the assembly code size as well as the number of memory operations produced by GCC using different optimization options, including "-O0", "-O1", "-O2", "-O3" and "-Os". We also compared the execution time in order to evaluate the run-time performance of these options.

This paper is organized as follows. Section 2 presents the motivation of our work and hypotheses. Section 3 describes experimental infrastructure. Section 4 describes experimental results and analysis. Section 5 contains conclusions, and section 6 shows limitations in our experiment and future work.

2. Motivation and Hypotheses

Computer pioneers correctly predicted that programmers would want unlimited amounts of fast memory. As shown in Figure 1, the performance gap between CPU and Memory has become increasingly larger.

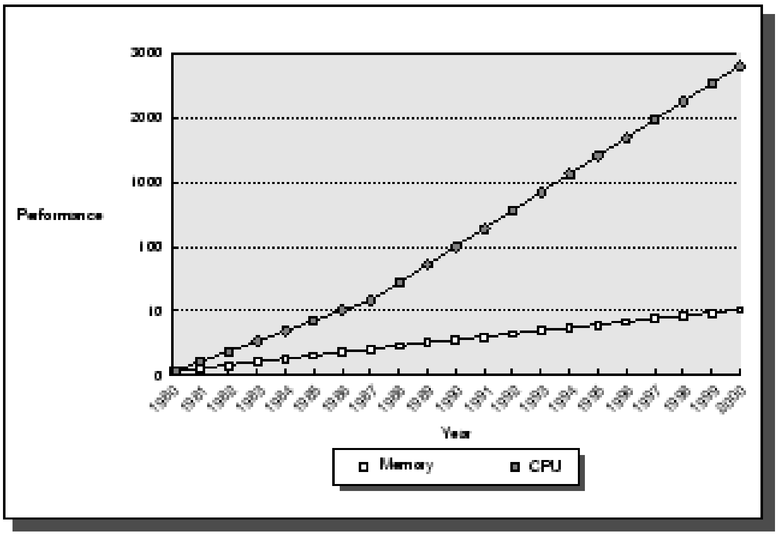


Figure 1. Starting with 1980 performance as a baseline,
the performance of memory and CPUs are plotted over time[1]

An economical solution to this gap is memory hierarchy. But from a compiler's perspective, given a computer architecture, part of the solution should be producing efficient code that can make full use of the memory hierarchy in that given architecture. Registers are the fastest locations in the memory hierarchy. Often, registers are the only memory locations most operations can access directly, that is, values should be loaded into registers before operation. Another fact is there are only a fix number of physical registers. For example[2], Intel 8086 has fourteen 16-bit registers. X86-64 architecture has sixteen 64bit general purpose registers, eight 64bit MMX registers, and sixteen 128bit SSE registers. Since the number of registers is limited, to transfer data, between memory and registers is certainly necessary. Certainly, this kind of operation is much less efficient than the operation between registers. To improve the efficiency is one goal of compiler, so it will make full use of every available register and reduce the number of memory operations. Besides, compiler will also take mechanisms to reduce code size and produce compact code.

Based on above motivation and facts, we propose following hypotheses:

(a) By using successive higher optimization options, GCC will reduce number of memory-related instructions in program.

(b) By using "-Os" (an option for code size optimization), the number of instructions must be the smallest when comparing with other optimization options.

(c) By using successive higher optimization options, GCC will generate more efficient executable file.

3. Experimental Infrastructure

3.1 Experimental environment

Our experiment was done on cy09@cs.arizona.edu and cy10@cs.arizona.edu. Both machines consist of 1GMHZ CPU of x86 architecture, running Linux Fedora Core Release 3 with 2GB memory. GCC version is 3.4.4 (20050721). We also used PLTO[4] (pronounced “pluto”) which is a Pentium Link-Time Optimizer developed by the department of Computer Science at University of Arizona.

3.2 Experimental procedure

The procedure of our experiment is shown in Figure 2. We would like to explain it in detail step by step.

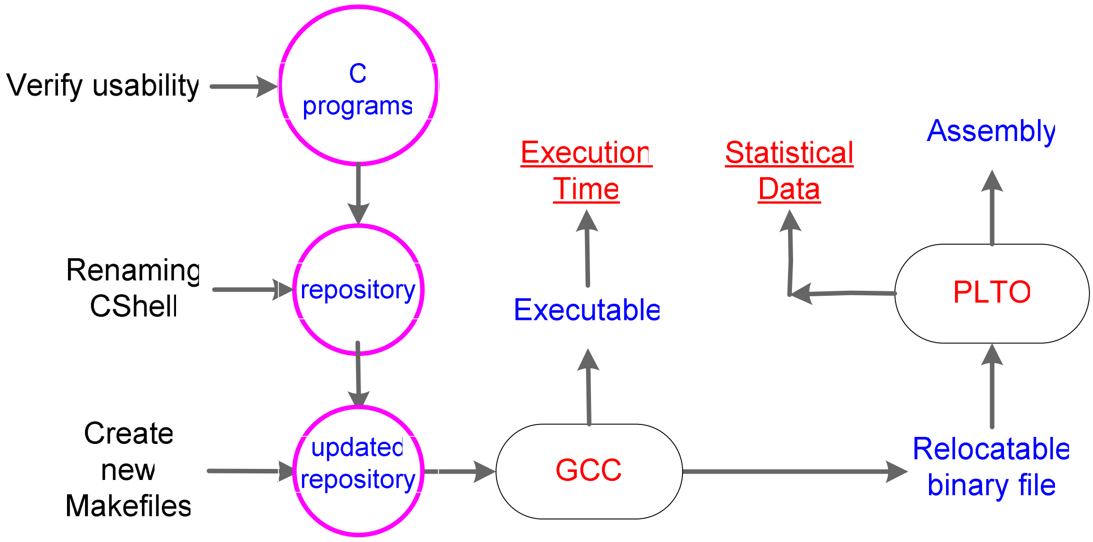


Figure 2 Experimental procedure

Step1. Verify usability of C program

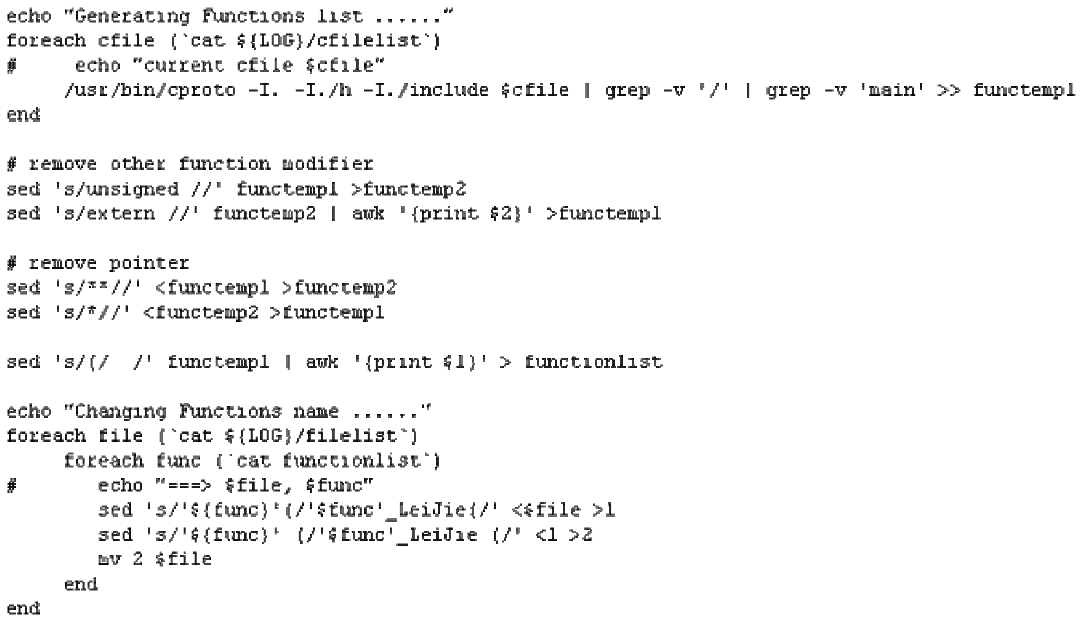
Given a C program, we firstly need to verify if this program can be used in our experiment. The verification is done in following way: compile this program and make sure it runs successfully in our environment; generate relocatable binary file using static link; run cproto (a tool used to generate all the function prototypes defined in the input source files) to get all functions defined in the program. If the program can pass all these steps, we put it into our program repository.

Step2. Renaming CShell

One important feature of PLTO is that it can only read a static-linked relocatable file. When generating static-linked relocatable file, GCC would link all specified library into binary, which means all the functions in the library will be linked into the relocatable file. Since we only care about the assembly code generated by our program, we need to “sift” other functions from the relocatable file.

One way to achieve this is to add a special postfix to all functions, including callers and callees, defined in our program. Then we modified PLTO source code and let it recognize the specific postfix we used. If the function contains the postfix, we count it. If no, we simply ignore it.

In order to rename all the function calls and callees, we wrote a C shell script. It calls “cproto” to generate the function prototypes list defined in the input source files, and then rename them. Figure 4 is the main part of the script.



```
echo "Generating Functions list ......"
foreach cfile (`cat ${LOG}/cfilelist`)
#     echo "current cfile $cfile"
     /usr/bin/cproto -I. -I./h -I./include $cfile | grep -v '/' | grep -v 'main' >> functempl
end

# remove other function modifier
sed 's/unsigned //' functempl >functemp2
sed 's/extern //' functemp2 | awk '{print $2}' >functempl

# remove pointer
sed 's/**//' <functempl >functemp2
sed 's/*//' <functemp2 >functempl

sed 's/(/  /' functempl | awk '{print $1}' > functionlist

echo "Changing Functions name ......"
foreach file (`cat ${LOG}/filelist`)
     foreach func (`cat functionlist`)
#        echo "===> $file, $func"
        sed 's/'${func}'(/'$func'_LeiJie(/' <$file >1
        sed 's/'${func}' (/'$func'_LeiJie (/' <1 >2
        mv 2 $file
     end
end
```

Figure 3 Renaming C shell script

Step3. Create new makefiles

New makefiles are created for two purposes: one is for generating executable for all optimization options; the other for generating relocatable file for all options.

To generate executable file, we just append "-O0", "-O1", "-O2", "-O3" and "-Os" to original compilation flag in the makefiles. To generate relocatable file, we should append "-static –Wl,-r" to compilation flag. For example, to generate a relocatable file for "-O1", we change compilation flag to be: "-O1 -static -Wl,-r".

After renaming and creating new makefiles, we got our new repository. We call it updated repository.

Step4. Run GCC to generate relocatable file and executable

This step simply calls makefiles created in step3. All relocatable and executable files will be generated automatically.

Step5. Run PLTO to get statistical data

Figure 4 is the PLTO framework. Given the relocatable file, PLTO will load it, dissemble the input binary file, create an instructions list, construct the control flow graph and then perform the optimizations. Since we are only interested the output of the GCC optimizations, we disabled the optimization functionality of PLTO.

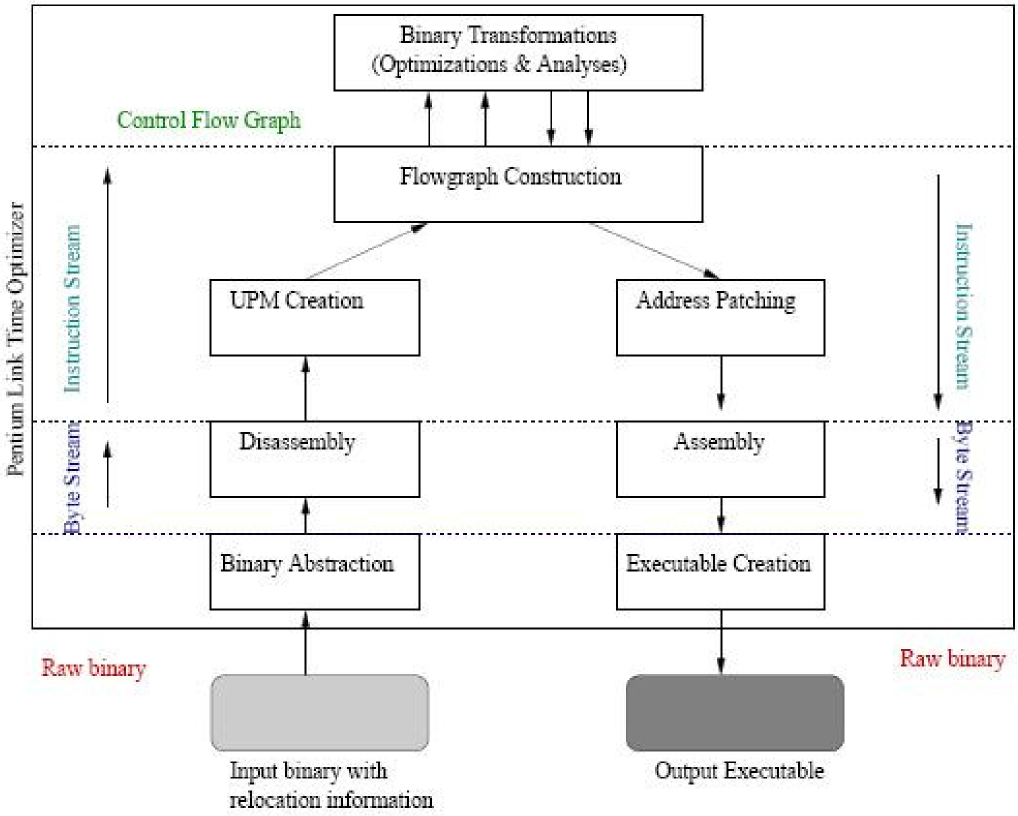


Figure 4 PLTO Framework[4]

3.3 Major changes in PLTO

We made three major changes in PLTO in order to make use of it in our experiment.

3.1 To recognize the postfix

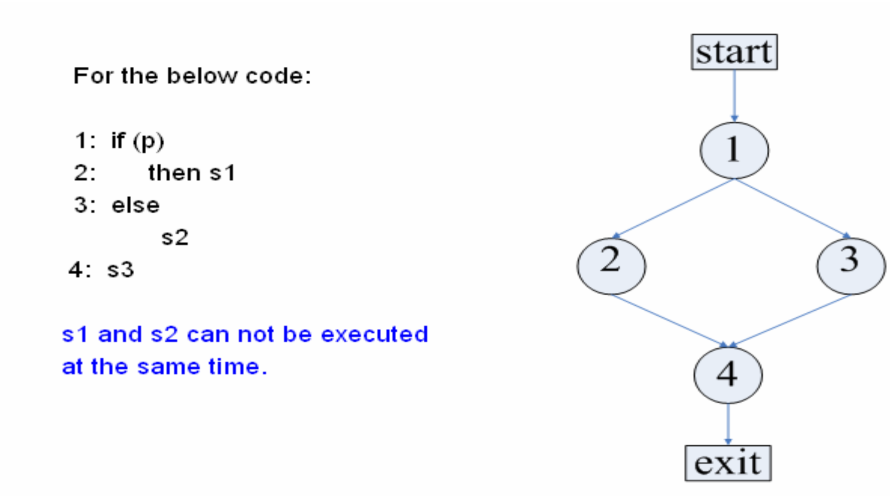
We have discussed this in section 3.2 experimental procedure step2.

3.2 Memory-related instruction calculation

We studied the x86 architecture instruction set documents and concluded the features of memory related instructions. We created a function named isMemoryInstr for this purpose. If format of an instruction satisfies the features of memory instructions, this function will return true. Actually, the calculation of memory-related instruction is done together with the total instruction calculation. We check each instruction, if isMemoryInstr returns true, then we increase the memory instruction counter.

3.3 Total instruction calculation

When calculating the total number of instructions, we can not simply add up all the instructions. We can easily understand this by following example described in Figure 4.



For the below code:

```
1:  if (p)
2:       then s1
3:  else
            s2
4:  s3
```

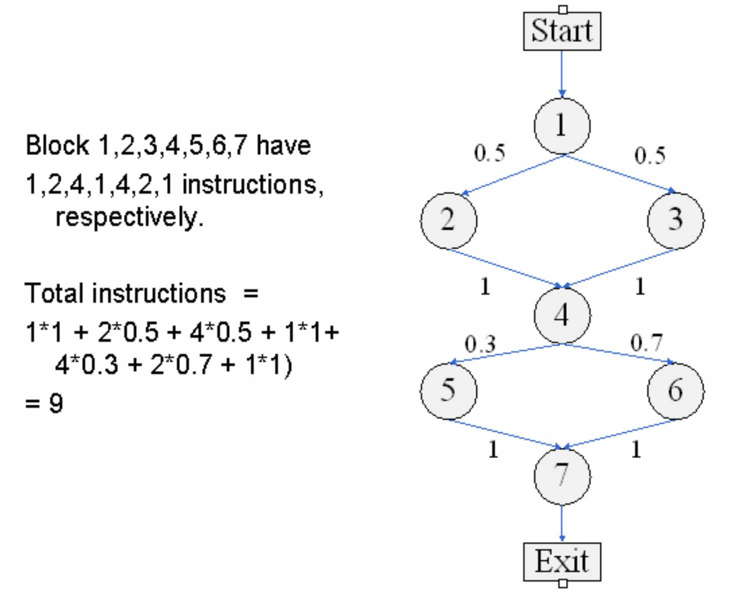
s1 and s2 can not be executed at the same time.

Figure 4 Branches in CFG

Obviously, s1 and s2 can not be executed at the same time, so we can not simply add the instructions in s1 and s2 together. In run-time, the program will always choose one from the two branches. And this choice usually can not be determined during the compile time. To get more accurate information about the number of instructions, we associate a weight to each basic block based on the probability of the block got executed. Since there is no way for us to get run-time information, we assume each branch of the same predecessor has the same opportunity to be taken.

If the weight of a basic block is 1, it means this block will be executed definitely. We first set the "start" weight of basic block to be 1. We assume that there is no loop in the CFG (Control Flow Graph). We would talk about how to deal with loop later. Then the weight of a basic block is the sum of weights of its predecessor(s) time(s) the probability of the CFG edge.

We use the CFG as well as dominator tree build by PLTO to get the predecessor and dominator information. More over, we added functions into PLTO to compute block weight, then calculate the number of instructions based on block weight. Figure 5 is an example on how to calculate the number of instructions when basic block weight is taken into account.



Block 1,2,3,4,5,6,7 have 1,2,4,1,4,2,1 instructions, respectively.

Total instructions = 1*1 + 2*0.5 + 4*0.5 + 1*1+ 4*0.3 + 2*0.7 + 1*1) = 9

Figure 5 Block Weight

Another issue we need to consider is the loop. Each loop is associated with a special edge called back edge. Based on the CFG and dominator information, we can easily know whether an edge is a back edge or not. Detail information about dominator information can be found in reference 2. If we find an edge is a back edge, we would mark this edge and remove it from CFG. To remove all the back edges in CFG is the basis of our previous solution.

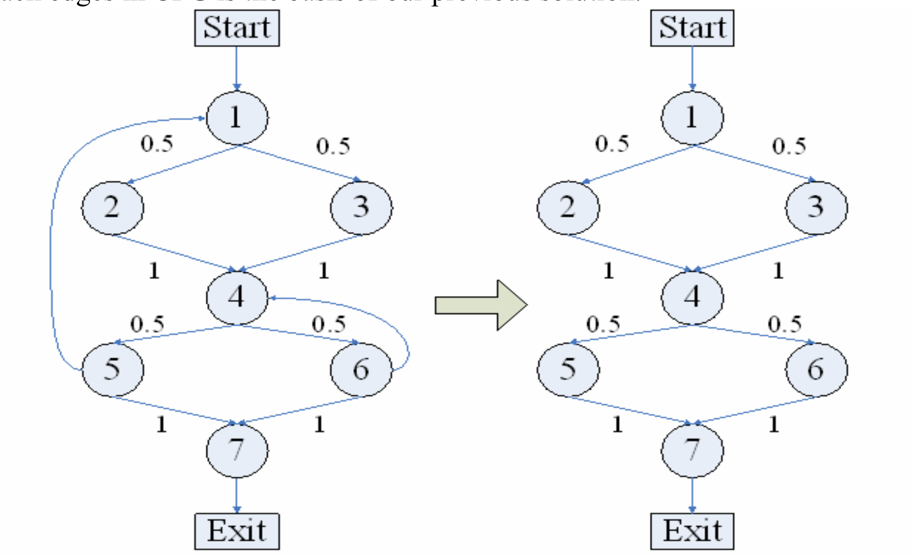


Figure 6 Remove back edge from CFG

However, the instructions in the loop might be executed many times, so we should consider how many times a loop would be executed when calculating instruction number. Unfortunately, we can not determine the how many iterations a loop is going to be executed during the compile time. Without any profile generated in run-time about the program, we assume each loop will be executed only once.

4. Experimental results and analysis

4.1 Total number of instructions

The total number of instructions of each program in our repository is shown in Figure 7. In x-dimension, all programs, numbered from 1 to 12, evolved in our experiment are listed. We counted the approximate line number of each program and ordered them. Y-dimension is the instruction number. We have following observations which worth are to mention:

4.1.1 -O0

The code size of "-O0" is always the largest in our experiment except program 7. All optimization options except "-O0" will make use of following common optimization techniques[5]. All of them can help reduce code size.

(1) Dead code elimination

This removes instructions which have no uses, thus reduces code size and eliminates unnecessary computation.

(2) Redundancy elimination

This reuses the results that are already computed, so the redundancy code can be removed.

(3) Common sub-expression elimination

This removes the duplicated sub-expression which also contributes to reduce code size.

(4) Constant folding and propagation

It replaces expressions consisting of constants with their final value at compile time, rather than doing the calculation in run-time.

Other optimization techniques can also reduce code size, but we don't list all of them in the paper. For more information, please refer to [5].

4.1.2 -Os

In almost all cases in our experiment, the code generated by "-Os" is the minimum within all optimization options. This fact consists with our hypothesis (b).

Interestingly, program 1 is an exception. As shows in Figure 7, the code size produced by "-Os" (192) is larger than "-O2" (177) and "-O3" (180). Though the difference in terms of number is small, it's enough to prove that "-Os" does not always generate smallest code size. And we also believe that this exception is not a single case.

4.1.3 -O3

Commonly, the code size of "-O3" is larger than that of "-O2". We finally figure out this phenomenon is mainly caused by "-finline-functions" sub-option. This switch is enabled in "-O3" internally. It will integrate all simple functions into their callers and result in code expansion.

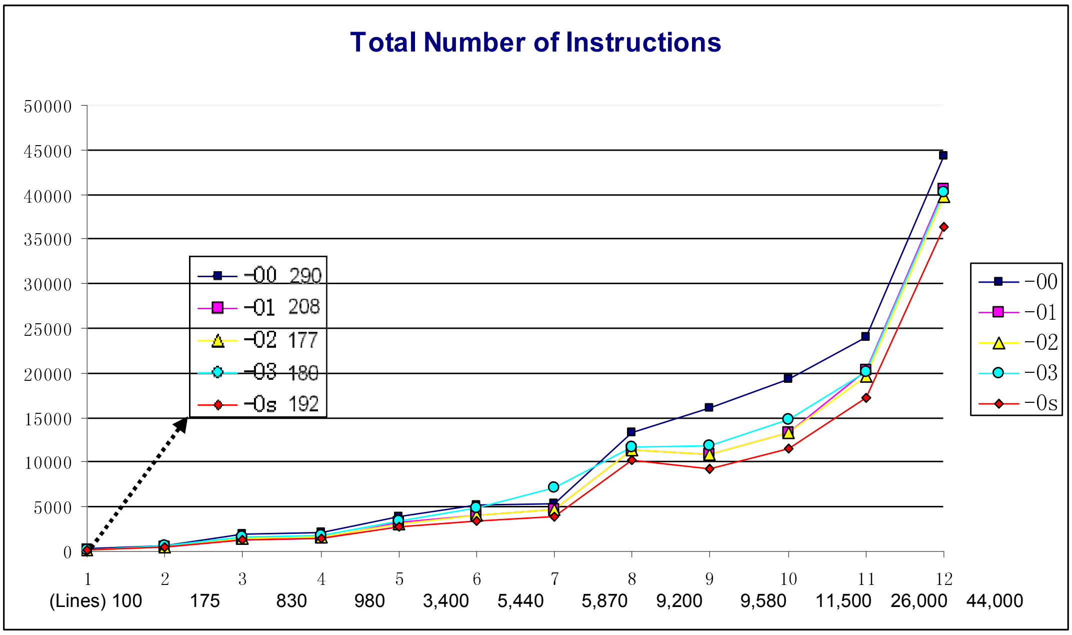


Figure 7 Total number of instructions

4.2 Memory-related number of instructions

As seen from Figure 8, the number of memory-related instructions in each option is basically in proportion the total number of instructions. In our experiment, the memory instructions count 37.50% to 65.84% of all instructions.

Parts of our results support our hypothesis (a), but others are not. We can see that the number of memory-related instructions of "-O0" is generally the largest, but it's can also smaller than "-O3", such as program 7. "-Os" is generally the smallest, but it's larger than "-O1" in program2 and 3. Similar to the total number of instructions, "-O3" is higher than "-O1" and "-O2" in most cases.

If the performance is directly determined by the total number and memory-related number, we can infer that the performance of "-O0" is the worst, while "-Os" is the best. Let's discuss this question further in next section.

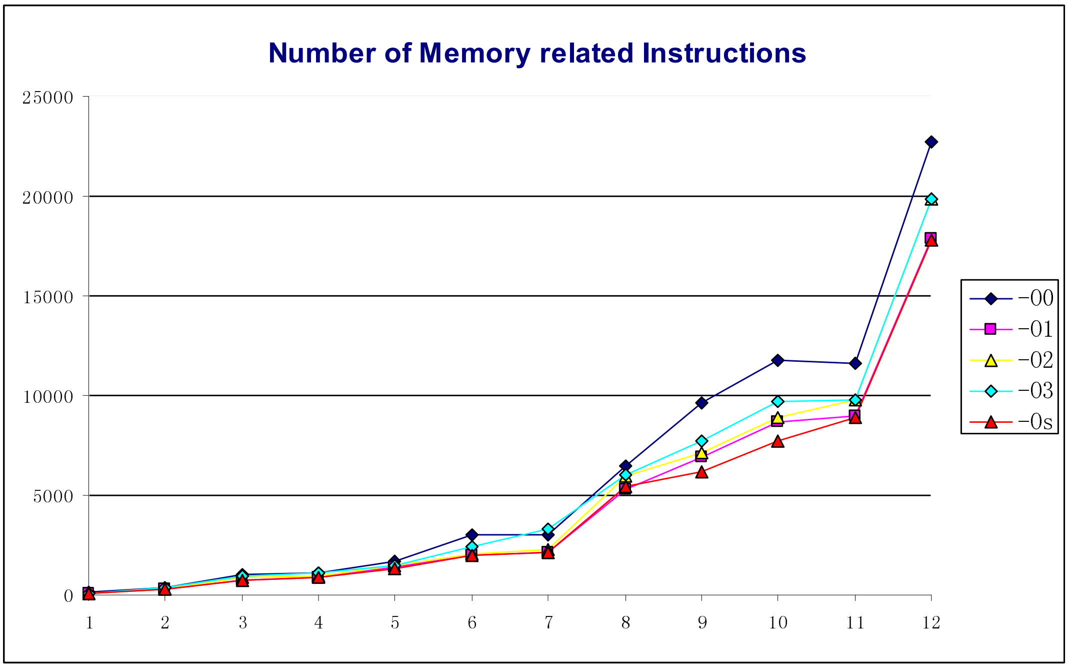


Figure 8 Number of memory-related instructions

4.3 Execution time

In order to evaluate hypothesis (c), we generated the executable for each non-interactive program and compare the execution time of different optimization options. As shown in Figure 9, we can see that hypothesis (c) is supported by the behavior of optimization options in program 1, 2 and 3. In fact, we have seen many other programs which consist with hypothesis (c).

However, Figure 10 to Figure 13 present unexpected phenomena which disprove hypothesis (c). In program 12 "-O1" performs worst, but "-Os" is the best in the same case. "-Os" is the worst in program 5. While in program 6 and 11, "-O3" is always the worst and "-Os" is the best. "-O0" is supposed to be the worst, but in all these cases, it's neither the worst, nor the best.

Two possible reasons may attribute to these interesting phenomena:

(1) Large I/Os

Program 5 is a page management program of Operating System and program 6, 11 and 12 are all Database related, such as index operations, queries and record operations. All of them require large I/Os in run-time. The I/Os, especially read I/Os, consume much CPU time. Whether the devices are busy or not while running the executable can play crucial role in experimental results.

(2) Inaccurate measurements

When we execute a binary file, we make use of "gettimeofday" function call to get the start and finish time. The difference between start and finish time is taken as the execution time of this binary file. Although, we keep monitoring the CPU and memory load during the tests and make sure that CPU load is lower than 40% and free memory is more than 50% while running the executable. Unfortunately, the process might be interrupted in run-time, so the execution time we got is not the actual time that the process possesses the CPU. Thus, the execution time is greater or equal to the real CPU time. In order to prevent such inaccuracy, we should count CPU time instead.

Considering the performance of “-O3”, it’s worse than “-O2” not in few cases. Sometimes, it’s even worse than other options. We consider it’s because “-O3” use too aggressive optimization mechanisms, some programs cannot benefit from them.

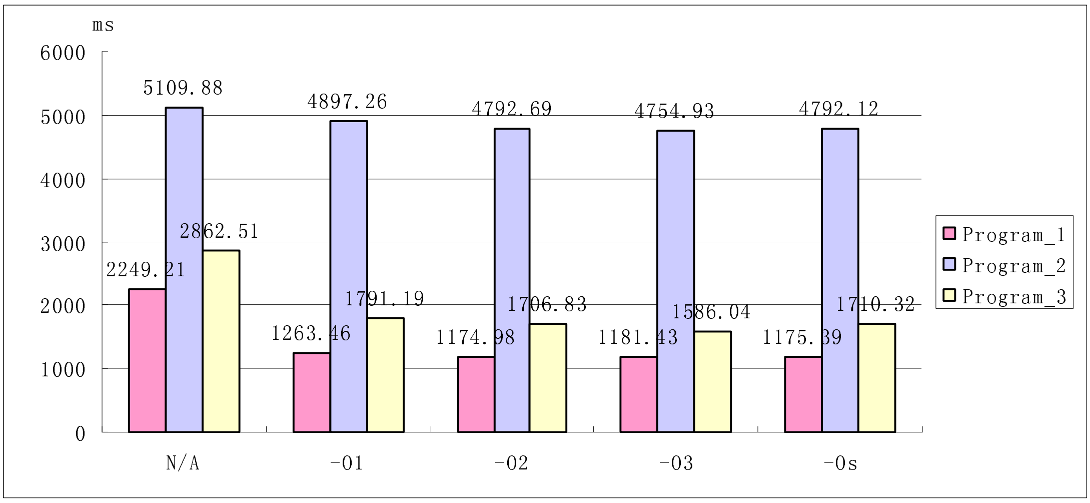


Figure 9

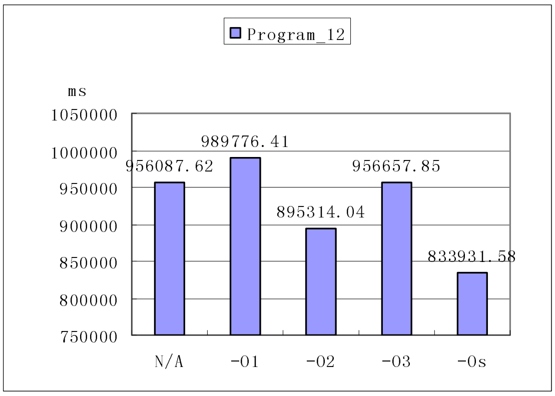


Figure 10

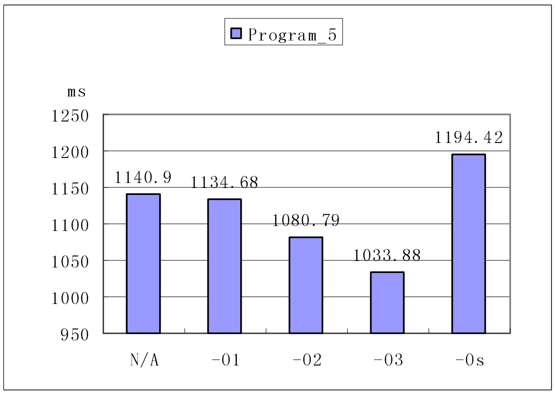


Figure 11

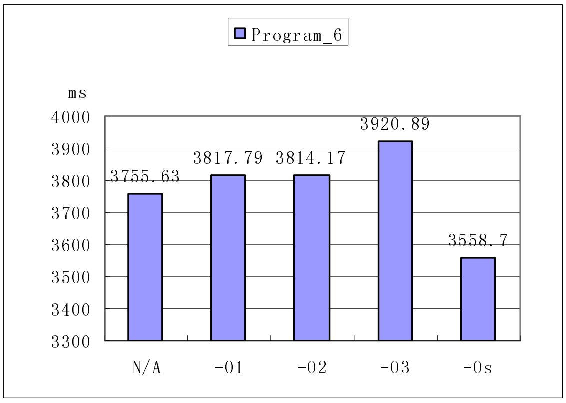


Figure 12

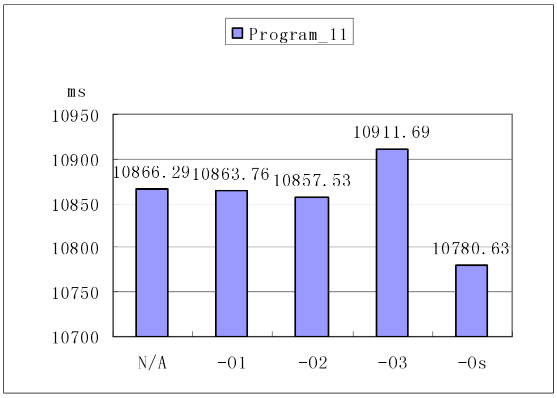


Figure 13

5. Conclusions

During the experiment, we found that our hypotheses are supported by most cases, but not all. It's worthy for us to evaluate these hypotheses and try to modify them based on the results we got for further study.

5.1 Evaluate hypotheses

(1) Hypothesis (a)
The facts that are conflicted with this hypothesis are various. In terms of memory-related instruction number, different order might occur in different program and we do not know the relationship between the order and nature of program.
(2) Hypothesis (b)
Currently, we haven't figure out the possible reasons that would account to the unexpected phenomenon in program 1, so we need more experiment before modifying it.
(3) Hypothesis (c)
We think the "unexpected" phenomenon relates to large I/Os, so we modified it as follows:
By using successive higher optimization options, GCC will generate more efficient executable when required number of I/Os is small.

5.2 Further discussion

Based on the whole study, we have new understanding on GCC:
(1) Successive higher optimization options can produce more efficient code in general, but GCC cannot guarantee this is always true.
(2) Consider the code size as well as memory operations, "-Os" normally generates the smallest code, but it might increase the size in certain cases. "-O3" is generally larger than "-O1", "-O2" and "-O3" because of function inline.

6. Limitations and future work

Although we made efforts to improve validity of our experiment, they still have several crucial limitations.

6.1 LOOP

When calculating the block weight, we simply ignore the loop by removing the back edge in CFG. In other words, we assume that the loop only executes once. If the number of loops can be know at compile time, we should take it into account.

6.2 Profile

If a node in CFG contains several branches, we assume that all these branches have the same possibility to be executed in run-time. But in reality, the possibility could be different. To be more accurate, we can estimate the possibility of each branch in run-time by simulation and store the information as profile. In compile time, we can read the profile and set possibility of each branch, then calculate the block weight. In this way, we can have more accurate estimation in block weight.

6.3 Cycle

Since we assume that the instructions are executed in sequence, that's the basis that the number of instructions can be used to estimate the performance. However, instructions are actually executed in parallel, such as software pipeline. Each instruction needs one or more cycles in order to execute. At the same time, several instructions can execute simultaneously in a single cycle. So if we can compute the total number of cycles needed in run-time, our experimental results would be more significant and meaningful.

We have some basic knowledge on GCC after this study, but our knowledge is far from enough. Hopefully, we can have opportunity to study it further and get a deep understanding of GCC.

Acknowledgments

We would like to thank Dr. Snodgrass for his contributions to our work. We would also thank Dr. Gupta and Dr. Debray for their precious suggestions. In particular, we want to thank HaiFeng He for his support. Finally, our work is based on PLTO developed by Computer Science department of U of Arizona.

[**References**]

1. John L. Hennessy, David A. Patterson, Computer Architecture: A Quantitative Approach, Third Edition (The Morgan Kaufmann Series in Computer Architecture and Design), 2002.
2. Brian J. Gough, An introduction to GCC – for the GNU compilers gcc and g++, 2005.
3. Muchnick, Steven S, Advanced compiler design and implementation, 1998.
4. Greg Andrew, Saumya Debray etc, The PLTO binary rewriting system - A programmer's manual , 2005.
5. Compiler optimization, http://en.wikipedia.org/wiki/Compiler_optimization, Wikipedia, 2007.
6. GCC manual page, 2005.