Advances in Computer Science and its Applications (ACSA)
Vol. 2, No. 3, 2013, ISSN 2166-2924

www.worldsciencepublisher.org

Application Performance Evaluation on Different Compiler Optimizations

Yiming Han

University of Texas at San Antonio, TX, USA

Email: yimingmining@gmail.com

Abstract –How to evaluate computer's performance is an important issue for engineers in the area of computer, especially for those vendors. Under different compiler optimizations, the application on the same computer performs diversely. In this project, we investigate the impact of application performance with different compiler optimization levels. We choose the compiler, GCC, as our target compiler. With the SPEC CPU2006 benchmark, we evaluate several benchmarks performance when compiling those with different optimization levels. In addition, we employ the metric tools, GCOV, to fetch the numeric data. Through the tables and figures, we analyze the impact of application performance affected by optimization levels.

Keywords – SPEC CPU2006, GCOV, GCC, Optimization

1. INTRODUCTION

The approaches to improve hardware performance, especially processor, have been studied and analyzed for many decades [10], [11]. In [1], it introduced many features, including deep pipelining, multilevel cache hierarchy, branch predictors, out of order execution engine, advanced floating point and multimedia units, aiming at improving the processor performance. However, to explore these features successfully, compilers adapting to specific architecture are needed. Such kind of compilers can do the "code efficient" job by understanding the hardware architecture. During the compile time, they optimize the source code, and generate the target-oriented executable codes, then do best to maximize the application performance with the computer architecture features.

Compiler optimization is the process by transforming the output of a compiler by minimize or maximize some effect/attribute of an executable program. It is very common used in modern compilers and indeed it achieves the better performance at the running time. To evaluate the effect of the compiler optimization, many metric parameters were introduced. The most common metric is the time taken to execute a program, and the other one is the amount of occupied memory when running the program [2].

Techniques to optimizing the program can be divided into many categories according to the different scopes, varying from a single statement, a basic block, several blocks, to the whole program. Those optimization scopes include loop optimizations, whole-program optimization(inter-procedural optimization), super-local optimizations and the local optimizations [2]. In each scope many optimization options are provided. Even when the same sources compiled by the same compiler with different optimization options and executed on the same computer, would display distinct performance. In 1994, [3] tried figured out that how these optimization options affect the application performance.

Hence, how to choose an appropriate optimization option becomes a question pointed by Kenneth H. in [4]. GCC has several standard optimization levels, such as -O0,-O1,-O2, and for each standard optimization levels there are numerous optimization options. It is infeasible to exhaust all the possible combinations of optimizations and find out the best one. Upon this viewpoint, some research on the relationship between the compiler performance and optimization options have grow up over the past years. In 2004, Swathi [1] found the relationship among optimizations of different compilers (GCC, ICC) and performance of the executable programs by studying the execution characteristics of SPEC CPU 2000 benchmarks suite using Intel's VTune Performance Analyzer on Linux platform.

In this paper, we adopt the SPEC CPU 2006 benchmark suite on different compiler optimization levels, and then characterize the static and dynamic behaviors of the benchmarks to study the performance of executable codes generated by compilers. To compare different compiler optimizations, we compile SPEC CPU 2006 benchmark suite with ICC and GCC on the windows operating systems. To characterize the behavior of programs, we collect performance information, analyze and present it visually.

The rest of this paper is organized as follows. Necessary preliminary knowledge is introduced in section

2 to help understand the whole experiment. Section 3 presents the numeric data collected during the experiments and comparisons. And the analysis is included. In section 4, the conclusion is drawn.

2. PRELIMINARY

In this section, we introduce the tools utilized in the experiments, including SPEC CPU2006, GCC, ICC, Vtune, and gcov.

2.1 SPEC CPU2006

In 2006, the Standard Performance Evaluation Corporation (SPEC) announced CPU2006 to replace the old version CPU2000. The SPEC CPU2006 benchmarks are widely used in both industry and academia and it provides a comparative measure of integer and/or floating point computing intensive performance. The SPEC CPU2006 benchmarks represent a wide range application area without similar characteristics programs. The programs are from real life applications, instead of artificial loop kernels or synthetic benchmarks. These benchmarks are provided as source code and require the user to be comfortable using compiler commands as well as other commands via a command interpreter using a console or command prompt window in order to generate executable binaries [5].

"SPEC CPU2006, combined performance of CPU, memory and compiler contains 12 SPEC2006 integer programs and 17 floating-point programs, including [5]:

- CINT2006 (”SPECint¨), testing integer arithmetic, with programs such as compilers, interpreters, word processors, chess programs etc.
- CFP2006 (”SPECfp”), testing floating point performance, with physical simulations, 3D graphics, image processing, computational chemistry etc.. "

2.2 GCC Compiler [8]

"The GNU Compiler Collection (GCC) is a compiler system produced by the GNU Project to support various programming languages. The original version of GCC could only handle the C programming language. However, now GCC has been extended to support many additional languages, e.g. Java, Fortran, ADA etc.. GCC has several significant features that make it the strongest free software compiler [8]:

- GCC is portable - it runs on most platforms available today and could produce output for many types of processors;
- `GCC could cross-compile any program, which means that it could produce executable files for a different system from the one used by itself;
- GCC has multiple language front-ends so that it could parse different languages;
- GCC has a scalable design, allowing support for new languages and architecture by adding new modules;
- Most importantly, GCC is a free software, meaning that anyone could use it and modify it under certain agreement.

GCC support a range of general optimization levels so as to control compilation-time, compiler memory usage, and the run-time speed and code scale. The optimization levels could be divided into three main standards, labeled by the number 0, 1, 2, together with individual options for specific types of optimization [8].

- -O0: This level does not provide any optimization and compiles the source code in the most straightforward way. This is the best way for debugging and is the default option if there is no optimization requirement.
- -O1: This level provides the most common forms of optimization which do not require any speed-space tradeoffs. Comparatively speaking, the executable files produced by this kind of optimization should be smaller and faster than with the first level - -O0, because of the reduced amounts of data that need to be processed after simple optimization.
- -O2: This option provides further optimization than the first two ones, since it could support certain levels of instruction scheduling. Based on this fact, the compiler takes longer to compile programs and needs further requirements for memory consumption than with -O1. And the executables should not increase in size.
- -O3: This option turns on more expensive optimizations, such as function inlining, in addition to all the optimizations of the lower levels -O2 and -O1. The -O3 optimization level may increase the speed of the resulting executable, but can also increase its size. Under some circumstances where these optimizations are not favorable, this option might actually make a program slower.
- -funroll-loops: This level aims at loop-unrolling which is independent of all the other three optimization options. It will increase the size of an executable. Whether or not this option produces a beneficial result has to be examined on a case-by-case basis."

2.3 ICC compiler [9]

"Intel C++ Compiler (ICC) is a collection of C and C++ compilers from Intel, which are available for Linux, Microsoft Windows and Mac OS X. Intel supports compilation for its IA-32, Intel 64, Itanium 2 processors and certain non-Intel but compatible processors, e.g., certain AMD processors. Intel C++ Compiler further supports both OpenMP 3.0 and automatic parallelization for symmetric multiprocessing.

Intel C++ Compiler belongs to the family of compilers with the Edison Design Group front-end. The compiler is

also notable for being widely used for SPEC CPU Benchmarks of IA-32, x86-64, and Itanium 2 architectures.
Intel's suite of compilers' front-ends support C, C++ and Fortran programming languages. And it has been compatible with GCC 3.2 and later releases.

Intel tunes its compilers to optimize for its hardware platforms to minimize stalls and to produce code that executes in the fewest number of cycles. The Intel C++ Compiler supports three separate high-level techniques for optimizing the compiled program: inter-procedural optimization (IPO), profile-guided optimization (PGO), and high-level optimization (HLO).

IPO applies typical compiler optimizations but using a broader scope that may include multiple procedures, multiple files, or the entire program, while the HLO are optimizations performed on a version of the program that more closely represents the source code. In HLO it includes loop interchange, loop fusion, loop unrolling, loop distribution, data pre-fetch, and more.

PGO could be seen as a mode of optimization where the compiler is able to access data from a sample run of the program across a representative input set. The data would indicate which areas of the program are executed more frequently, and which areas are executed less frequently. All optimizations benefit from profile-guided feedback because they are less reliant on heuristics when making compilation decisions.

Additionally, IPO may also include typical compiler optimizations on a whole-program level, for example dead code elimination, which removes code that is never executed. To accomplish this, the compiler tests for branches that are never taken and removes the code in that branch."

2.4 GCOV [12]

GCOV is a test coverage program. Together with GCC, it can help programmers to analyze the source code to create more efficient code and find untested parts of program. As a profiling tool, GCOV can help to discover which optimization can best affect the source code. In addition, GCOV can work with other profiling tool, e.g gprof, to access which parts of codes use the greatest amount of computing time. In the experiment, GCOV will help to gather the some metrics, including block number of the whole program, reachable block number and block number of execution number larger than 1000. For the detail of how to use the GCOV, please refer to
[12]. In this experiment, we use the optimization parameters: -O2 -fprofile-arcs -ftest-coverage; -O2 defining the optimization level.

3. EXPERIMENT RESULTS AND ANALYSIS

In this section, we present the experiment results. We show those numeric data sets with the tables and figures, and analyze those data sets.

3.1 Selected Benchmark

For SPEC CPU2006, the following integer and floating point benchmarks are selected:

- 401.bzip2
- 403.gcc
- 429.mcf
- 433.milc
- 444.namd
- 447.dealII
- 450.soplex
- 456.hmmer
- 458.sjeng
- 473.astar

3.2 Hardware Setting

In this experiment, we have three platforms to run these benchmarks. The hardware settings are listed in TABLE I.

OS	CPU	L1 Cache	L2 Cache	Memory
WIN7	Intel P8600 2.4GHZ	32KB + 32KB	3MB	2GB

TABLE I HARDWARE SETTING

3.2 Result

Because of that GCOV only works together with GCC to fetch the coverage information, so the metric for the GCC and ICC is not the same. Therefore, in the following we will display two different results of benchmarks on the different compilers.

1) Static metrics: In the experiment, the static metrics include:

- The number of basic blocks in the whole program;
- The number of basic blocks reached when running the program
- The number of basic blocks that were executed more than 1000 times

Since not all the benchmarks can get static metrics listed above. In the experiment, only the benchmarks, 401.zip2,
403. gcc, 429.mcf,433.milc,456.hmmer and 458.sjeng, can result in the amount of basic blocks. Also, GCOV only can work together with GCC. Hence, the following static metric are fetched upon the platform P1. For optimization level O0, O1, O2 and O3, the static metrics are presented in the following Figures 1, 2, 3 and 4.

Benchmarks	# of BB	# of reached BB	# of BB executed > 1000
401.zip2	2455	1551	1141
403.gcc	156425	54081	49471
429.mcf	424	325	244
433.milc	3276	1332	635
456.hmmer	10119	1685	484
458.sjeng	5091	2373	1916

Fig 1. Static metrics in optimization level O0

Benchmarks	# of BB	# of reached BB	# of BB executed > 1000
401.zip2	2150	1326	966
403.gcc	140312	59102	58001
429.mcf	386	293	212
433.milc	2853	1105	461
456.hmmer	8988	1518	393
458.sjeng	4571	2150	1727

Fig 2. Static metrics in optimization level O1

Benchmarks	# of BB	# of reached BB	# of BB executed > 1000
401.zip2	2215	1411	1029
403.gcc	140312	59102	58001
429.mcf	406	304	213
433.milc	3044	1212	496
456.hmmer	9376	1567	417
458.sjeng	4679	2170	1735

Fig 3. Static metrics in optimization level O2

Benchmarks	# of BB	# of reached BB	# of BB executed > 1000
401.zip2	2445	1402	1027
403.gcc	156425	54081	49471
429.mcf	437	314	224
433.milc	3596	1393	586
456.hmmer	10756	1692	436
458.sjeng	5387	2388	1944

Fig 4. Static metrics in optimization level O3

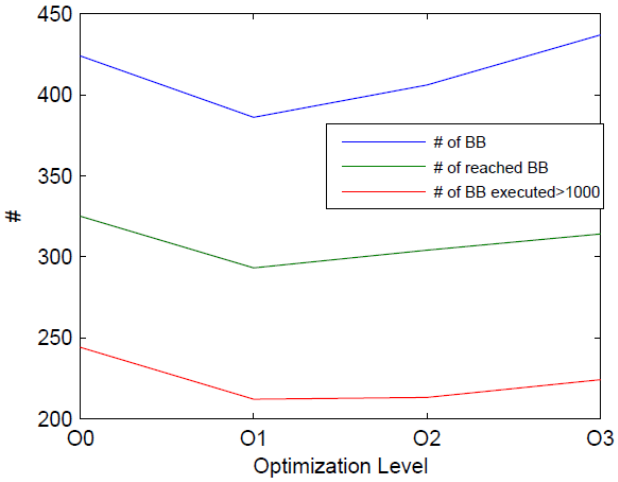

Fig 5. static metrics for 429.mcf with different optimization level

Take the benchmark 429.mcf as example. By plotting the amount of the basic blocks, the amount of reached basic blocks and the amount of basic blocks executed more than 1000 time in the Figure 5, it shows that by utilizing the optimization, those numbers are not always decreased. The numbers in the optimization level O1 show less than the optimization level O0. However, the numbers in the optimization level O3 is greater than those in the optimization level O2 and similar to O0.

For those benchmarks, let us look at the mean value of for each optimization levels. Define that

$$\overline{\#ofBB} = (\sum \#\ ofBB\ for\ all\ benchmarks)/6$$

Also, we can define the metrics $\overline{\#ofreachedBB}$ and $\overline{\#ofBB\ executed > 1000}$.

Figure 6 shows the average number of blocks in different optimization levels. As can be seen, the number of basic blocks in the optimization level O0 is larger than others. Because O0 does not support any optimizations and it does not eliminate any redundancies of the source code. At O1 optimization level, the compiler does some redundancy elimination and optimization jobs like constant propagation, lazy code motion, useless elimination etc.. O2 and O3 offer more efficient optimizations of the source code but they trade space for time, like loop unrolling. Thus they need more number of blocks and space to realize high level optimizations.

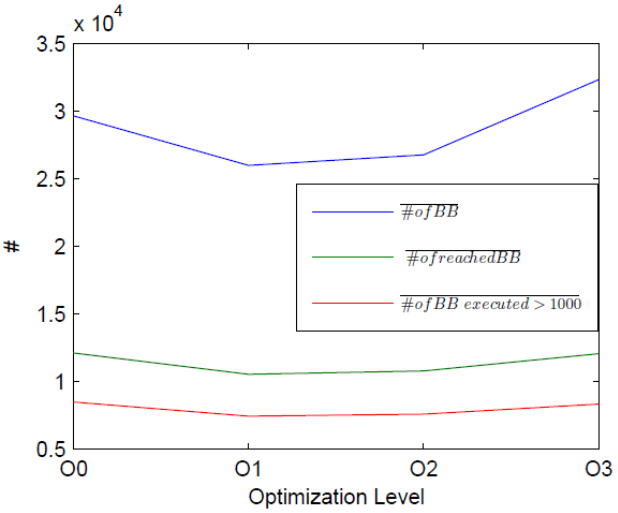

Fig 6. Mean value of static metrics for all benchmarks with different optimization level

Another static metric for the benchmarks is the size of executable program. When the compiler applies distinct optimization levels, the size of executable program varies. Sizes of all executable programs for each optimization level are listed in Figure 7.

Benchmark	O0	O1	O2	O3
401.bzip2	166,703	143,146	143,046	155,779
403.gcc	6,324,580	5,575,614	5,712,521	5,900,911
429.mcf	75,606	73,909	74,933	75,957
433.milc	264,626	246,192	256,536	283,672
444.namd	469,630	325,229	338,300	336,764
447.dealII	9,518,670	7,309,882	6,531,131	6,569,222
450.soplex	1,117,067	990,400	962,791	989,080
456.hmmer	555,595	487,273	498,979	536,517
458.sjeng	281,930	260,087	265,271	282,765
473.astar	106,091	92,036	89,490	93,586

Fig 7. Size of all executable programs for each optimization level.

From Figure 7, we can see that by using the optimizations, the size of executable program will decrease. However, it is not all the case for all the optimization levels. For the optimization level O3, the sizes of executable programs for all benchmarks are all greater than those upon the lower optimization level O2.

Indeed, such trends reflect the variations of the numbers of the basic blocks displayed in the Figures 1-4.

2) Dynamic metrics

Execution time is one of the most important metric to evaluate the influence of the optimizations. In the experiment, we run the above benchmarks upon different optimization levels. In the SPEC CPU 2006, the iteration of execution is 3 by default, but we set the iteration equaling to 1 to save the running time. All the benchmarks' execution times are listed in Figure 8.

Benchmark	O0	O1	O2	O3
401.bzip2	1532	1420	1410	1410
403.gcc	1310	1110	1041	969
429.mcf	963	853	848	733
433.milc	1287	1190	1170	1140
444.namd	1038	959	940	928
447.dealII	1361	1270	968	961
450.soplex	984	889	809	783
456.hmmer	1941	1870	1750	1689
458.sjeng	1691	1640	1490	1401
473.astar	1131	1020	1020	978

Fig 8. Execution time (seconds) of all executable programs for different optimization levels.

Figure 8 and Figure 9 show the general trends of execution speed (elapsed run-time) under four different optimization levels for the ten selected benchmarks. From the data, we could see that the execution time is based on decreased tendency. The -O0 level does nothing in any optimization and only compiles the programs in the most straightforward way so that the executable files under this level take the longest run time; The -O1 level of optimization does limited work in data reduction, thus it takes less time than the -O0 level comparatively; As referred to the next level, the collected run-time data indicate its further-level optimization obviously (theoretically, this option supports instruction scheduling which makes the compiling time longer and needs more memory space but makes the executable files running faster); The last column in this table contains the -O3 level results which are less than the three previous levels in most cases (the real data we collected in our experiments show two identical second counts in 401.bzip2) because of certain more expensive optimizations, e.g. function inlining.

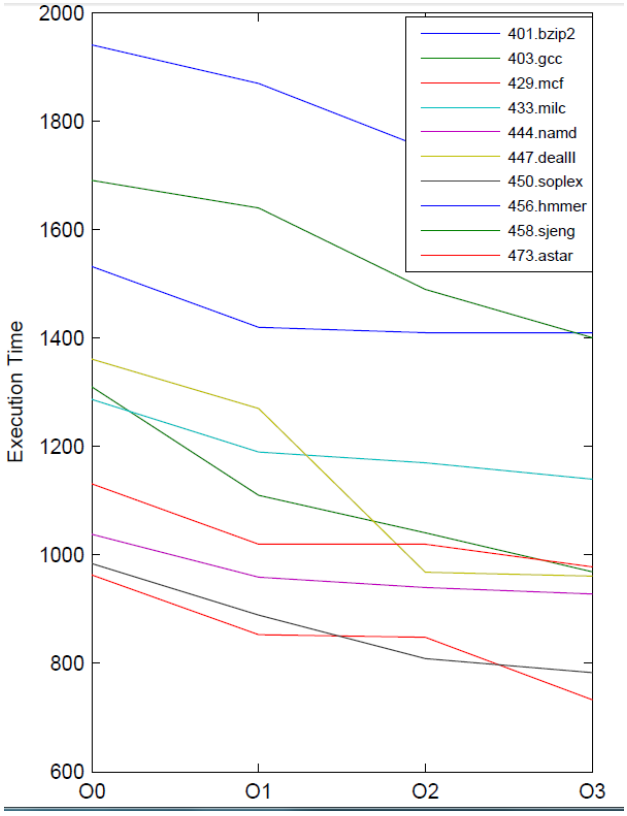

Fig 9. Plot for Execution time of all executable programs for different optimization levels.

4. CONCLUSION AND FUTURE WORK

In this paper, we studied how to use the benchmark suite and relevant applications to explore the application performance and interesting features. We utilized GCC, SPEC CPU2006 and GCOV to gather static and dynamic metrics, analyzed the influence upon different optimization levels. With the metrics listed above, maybe the programmer can determine which optimization level is better when considering the issues, such as code size, execute time, etc.. Cloud computing [22] - [27] is emerging as a powerful technology to meet the requirements for high-performance computing and massive storage. The performance [13] - [16] and security [17] - [21] are big issues on cloud systems. In the future, we will study the performance on cloud systems.

References

[1] Swathi Tanjore G., Aleksandar M., Execution Characteristics of SPEC CPU2000 Benchmarks: Intel C++ vs. Microsoft VC++, *ACM SE' 04*, 2004

[2] http://en.wikipedia.org/wiki/Compiler optimization

[3] Stewart, K. E., and White, S.W. The Effects of Compiler Options on Application Performance. In *Proceedings of IEEE International Conference on*

Computer Design: VLSI in Computers and Processors, (ICCD '94), 1994,340-343.

[4] Kenneth H., Lieven E., COLE: Compiler Optimization Level Exploration, *International Symposium on Code Generation and Optimization (CGO) '08*, 2008

[5] www.sepc.org/cpu2006

[6] http://en.wikipedia.org/wiki/SPECint

[7] http://software.intel.com/en-us/intel-vtune/

[8] Brian Cough, An introduction to GCC for the GNU Compilers gcc and g++, Feb, 2004.

[9]http://software.intel.com/en-us/forums/intel-c-compiler/

[10] Brian Fahs, Todd Rafacz, Sanjay J. Patel and Steven S. Lumetta, Continuous Optimization, *Proceedings of the 32nd annual international symposium on Computer Architecture*, 2005,86-97.

[11] C. Cascaval, E. Duesterwald, P. F. Sweeney and R. W. Wisniewski, Performance and environment monitoring for continuous program optimization, *IBM Journal of Research and Development,*Volume 50, 2006, 239 - 248.

[12] http://gcc.gnu.org/onlinedocs/gcc/Gcov-Intro.html

[13]Yiming Han and Anthony T. Chronopoulos, Distribut ed Loop Scheduling Schemes for Cloud Systems, *2013 IE EE 27th International Parallel and Distributed Processin g Symposium Workshops & PhD Forum (IPDPSW)*, Bosto n, MA, May 2013.

[14] Yming Han and Anthony T. Chronopoulos, Scalable Loop Self-Scheduling Schemes Implemented on Large-Scale Clusters, *2013 IEEE 27th International Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW)*, Boston, MA, May 2013.

[15] Yming Han and Anthony T. Chronopoulos, A Hierarchical Distributed Loop Self-Scheduling Scheme for Cloud Systems, *The 12th IEEE International Symposium on Network Computing and Applications, NCA 2013*, Boston, MA, August 2013.

[16] Zhou, Zihao, Feng Gao, and D. Wayne Goodman. Deposition of metal clusters on single-layer graphene/Ru (0001): Factors that govern cluster growth, *Surface Science*, 604.13 (2010): L31-L38.

[17] Qingji Zheng and Shouhuai Xu. 2012. Secure and efficient proof of storage with deduplication. In *Proceedings of the second ACM conference on Data and Application Security and Privacy*(CODASPY '12). ACM, New York, NY, USA, 1-12.

[18] Qingji Zheng and Shouhuai Xu. 2011. Fair and dynamic proofs of retrievability. In *Proceedings of the first ACM conference on Data and application security and privacy* (CODASPY '11). ACM, New York, NY, USA, 237-248.

[19] Qingji Zheng, Shouhuai Xu, and Giuseppe Ateniese. 2012. Efficient query integrity for outsourced dynamic databases. In *Proceedings of the 2012 ACM Workshop on Cloud computing security workshop* (CCSW '12). ACM, New York, NY, USA, 71-82.

[20] Xiangxue Li, Qingji Zheng, Haifeng Qian, Dong Zheng and Jianhua Li, Toward optimizing cauchy matrix for cauchy reed-solomon code, *IEEE Communications Letters*, vol.13, pp.603,605, August 2009

[21] Qingji Zheng; Xiangxue Li; Dong Zheng; Baoan Guo, Regular Quasi-cyclic LDPC Codes with Girth 6 from Prime Fields, *2010 Sixth International Conference on Intelligent Information Hiding and Multimedia Signal Processing (IIH-MSP)*, Oct. 2010

[22] Hangwei Qian, Chenghua Cao, Li Liu, Hualong Zu, Qixin Wang, Menghui Li, Tao Lin, Exploring the Network Scale-out in virtualized Servers, *Proceeding of International Conference on Soft Computing and Software Engineering (SCSE 2013).*

[23] Qixin Wang, Chenghua Cao, Menghui Li, Hualong Zu (2013) A New Model Based on Grey Theory and Neural Network Algorithm for Evaluation of AIDS Clinical Trial, *Advances in Computational Mathematics and its Applications, Vol.2, No.3, PP. 292-297.*

[24] Hualong Zu, Qixin Wang, Mingzhi Dong,Liwei Ma, Liang Yin, Yanhui Yang(2012), Compressed Sensing Based Fixed-Point DCT Image Encoding, *Advances in Computational Mathematics and its Applications, Vol.2, No.2, PP. 259-262.*

[25] Hangwei Qian, Hualong Zu, Chenghua Cao, Qixin Wang (2013), CSS: Facilitate the Cloud Service Selection in IaaS Platforms, *Proceeding of IEEE International Conference on Collaboration Technologies and Systems (CTS).*

[26] Hangwei Qian, Qixin Wang, (2013) Towards Proximity-aware Application Deployment in Geo-distributed Clouds, *Advances in Computer Science and its Applications, Vol.2, No.3, PP. 382-386.*

[27] Qixin Wang, Yang Liu, Xiaochuan Pan (2008), Atmosphere pollutants and mortality rate of respiratory diseases in Beijing, *Science of the Total Environment,* Vol.391 No.1, pp143–148.