AN EFFECTIVE ORCHESTRATION ALGORITHMS PERTAINED TO BENCHMARK APPLICATIONS

Andrews J.[1], **Sasikala T.** [2]

[1]Research Scholar, Sathyabama University, Chennai, India
[2]Principal, SRR Engineering College, Padur, Chennai, India
Email: [1]andrews_593@yahoo.com

ABSTRACT

Code optimization involves the application of rules and algorithms to program code, and its main objective is to run the code faster with lesser memory. But achieving this target involves lot of complication because arriving at the compiler configuration for a particular problem is a complex process. The performance of the program measured by time and memory depends on the machine architecture, problem domain and the settings of the compiler. There have been several proposed techniques that search the space of compiler options to find solutions. However such approach can be expensive. In current compilers, through command line arguments, the user must decide which optimization is to be applied in a given compilation run. But it is not a long term solution. Because compiler optimizations get increasingly numerous and complex, this problem must find an automated solution.In this paper, it is proposed to study the classification of problems, identification of ideal objective functions for different tasks and the ordering of objective function for optimization. In this paper we proposed an effective orchestration algorithm to select best set for a particular problem from larger set options. Many previous works consider only limited set of options. In this paper, we implemented compiler optimization selection algorithms such as branch and bound strategy and advanced combined elimination algorithm and evaluated its execution speed up. We argue that advanced combined elimination algorithm works better when compared to branch and bound strategy by showing experimental results using benchmark applications.

Key words: Compiler optimization, Benchmark Applications, Branch and Bound, Combined Elimination

I. INTRODUCTION

Recent version of compiler provides a larger set of optimization techniques, for users to fine tune the performance of various benchmark applications. But selecting the best set of options is not an easy, task especially for average users who do not have in depth understanding of the compiler options. Because selecting best set of options from various candidate options depends on system architecture and problem domain. Each optimization tries to improve the performance of the application, although they are not always effective .Through command line flags, the user can decide which optimizations are to be applied in a given compilation run, but clearly it is not a long term solution. As compiler options get increasingly numerous and complex, the problem must find an automated solution.

Because modern compiler provides larger number of optimization techniques and complex interactions among its various techniques finding optimal functions for specific benchmark applications is hard and time consuming process. There has been much previous work on automatically searching for the best optimization settings. This previous work is based on iteratively enabling certain optimizations, running the compiled program and, based on its performance, deciding on a new optimization setting. Pan *et al* introduce a new algorithm called *combined Elimination* (CE) [2],[3] that was shown to outperform all previous search-based techniques in finding good optimization settings with considerably fewer evaluations. However, these pure search or “orchestration” approaches do not use prior knowledge of the hardware, compiler, or program and instead attempt to obtain this knowledge online. Thus, every time a new program is optimized, the system starts with no prior knowledge.

In this paper we have considered modified version of combined elimination algorithm. This algorithm is compared with branch and bound strategy.GCC compiler provides three levels of optimization techniques [1],[5-6]. To obtain the best performance a user usually applies the highest optimization level –O3.In this level the compiler perform the most extensive code analysis and expects the compiler generated code to deliver the highest performance. In this paper we proposed effective orchestration algorithm to select the compiler options for a particular problem from large set options. Many previous works consider only limited set of options. For this work, we implemented compiler optimization selection algorithm advanced combined elimination can decide which optimizations are to be applied in a given compilation run, but clearly it is not a long term solution. As compiler options get increasingly numerous and complex, the problem must find an automated solution.

II. ORCHESTRATION ALGORITHMS

2.1 Advanced combined elimination strategy

Let S be the Set of available Optimization options

Let B represents selected compiler options set.

(i) Find T_B, by applying all flags are ON.

(ii) Compile the program with T_B configuration and measure the program performance.

(iii) Calculate Relative improvement percentage (RIP) for each and every optimization options. Relative improvement percentage is calculated based on finding the time required by applying particular flag ON and OFF with respect to T_B.

(iv) Store all the values in an array based on ascending order. i.e the most negative RIP is stored in first position of the array.

(v) Remove the first two most negative RIP's from an array instead of one. Now the value of T_B is changed in this step.

(vi) Remaining values in an array i.e (i) vary from 3 to n, Calculate RIP, and store the negative RIP's in array.

(v) If all values in an array represent positive values then set of flags in B represents best set.

Else

(vi) Repeat steps (ii) until B contains only positive values.

(vii) Stop

2.2. Branch and Bound Strategy Algorithm

Let S be the Set of available Optimization functions such as { F1, F2, … Fn }

Let B be the set having appropriate settings which is either ON or OFF

(i) Initialization.

(ii) Initialize default individual setting of flag for each function in set B

(iii) Calculate RIP_UB

(iv) While (s<> empty) repeat step (v)

(v) For each Fx in S, x=1 to n

Compute RIP (Fx=0)

Find A,the option in S with most negative RIP.

Find the subset *S′* from S, of those functions whose RIP<

(50% of RIP_UB)

if *S′* set is empty then return;

else

For all the elements in *S′*

Set respective flag off in B; Remove it from S

if RIP (F_A=0)<RIP_UB

RIP_UB=RIP (F_A=0)

(vi) Result Ready;

(vii) Stop

III. EXPERIMENTAL SETUP

In this paper we have considered GCC Version (4.3.2). GCC provides three levels of optimization techniques. Previous work considered only limited set of optimization techniques. This paper proposes larger set of optimization techniques.

3.1 Levels of Optimization

LEVEL – "O0"

With this level of optimization the compiler tries to reduce the code size and execution time without performing any optimization.

LEVEL – "O1"

Optimizing compilation takes more time and more memory for a large function.

LEVEL – "O2"

This level optimizes more than the previous level.

LEVEL – "O3"

This option turns on more expensive optimizations such as function in lining, in addition to all the optimizations of the lower levels. This optimization may increase speed of the resulting executable.

3.2 Benchmark Applications

The Mibench benchmark [4] suite programs were used to experiment the proposed algorithm. These benchmark suites are comparable with SPEC benchmark suite.

1. Bzip2: for compression
2. Consumer_jpeg_c: To add annotations, titles, index terms, etc. in JPEG files.
3. Consumer_tiff2bw: converts an RGB to a grayscale image by combining percentages of the red, green and blue channels.
4. qsort: for performing sorting.
5. dijkstra: for computing shortest paths.
6. patricia: datastructure used in place of full trees with very sparse leaf nodes.
7. security blowfish: document encoding and decoding.
8. SUSAN: feature detection.

3.3 Metrics used for Evaluation

Relative Improvement Percentage (RIP), RIP (Fi), which is the relative difference of the execution times of the two versions with and without Fi.

$$RIP\ (Fi)=T(Fi=0)-T(Fi=1)/T(Fi=1)\times 100 \qquad [1]$$

If Fi=1 then Fi is ON, else OFF

The baseline of this approach switches on all optimizations.

$T_B=T(Fi=1)=T(F1=1,F2=1,\ldots Fn=1)$, Where T_B represents base time.

$$RIP\ (Fi=0)=T(Fi=0)-\ T_B/T_B\times 100 \qquad [2]$$

If RIP (Fi=0) <0, the optimization of Fi has a negative effect, so it is better to turn off the function.

Architecture used for testing was Intel Core 2 Duo processor at 2.80 Ghz. With 4GB RAM using ubuntu operating system with 2MB L2 cache, and the compiler used for testing was GCC 4.3.2.

3.4 Creation of script file for automation

```
#!/bin/sh
Loc=/nn/qsort.c
#log file description
Log1=/nn/first.log
Log2=/nn/res.log
Log3=/nn/res2.log
Log4=/nn/res3.log
Log5=/nn/res4.log
#Level-"o1"
gcc –Wall -o1 –fauto-inc-dec $loc –o dump1
gcc –Wall -o1 –fno-auto-inc-dec $loc –o dump2
gcc –Wall -o1 –fcprop-registers $loc –o dump3
gcc –Wall -o1 –fno-cprop-registers $loc –o dump4
gcc –Wall -o1 –fdce $loc –o dump5
gcc –Wall -o1 –fno-dce $loc –o dump6
#Level-"o2"
gcc –Wall -o2 –falign-labels $loc –o dump7
gcc –Wall -o2 –fno-align-labels $loc –o dump8
gcc –Wall -o2 –falign-loops $loc –o dump9
gcc –Wall -o2 –fno-align-loops $loc –o dump10
#Level-"o3"
gcc –Wall -o3 –finline-functions $loc –o dump11
gcc –Wall -o3 –finline-functions $loc –o dump12
gcc –Wall -o3 –funswitch-loops- $loc –o dump13
gcc –Wall -o3 –fno-unswitch-loops- $loc –o dump13
#writing flag ON time in a file
/usr/bin/time –f "\t\t\t %e" –o $Log1 /nn/.dump1
while read line;
do
echo –e "$line";
file1=$line;
done<$log1
cat /dev/null>$log1
```

IV. RESULTS AND DISCUSSIONS

Using branch and bound optimization algorithm we have selected best set of optimal techniques for a given benchmark applications after much iteration.

Execution time is measured for each and every bench mark applications. Similarly with the help of advanced combined elimination algorithm we have selected best set of optimal techniques. Then these two algorithms are compared based on their execution speed up. The following Table 1 shows execution speed up between combined elimination algorithm and branch and bound strategy.

Table 1. Execution speed up

Benchmark	Advanced combined elimination	Branch and bound
Bzip 2	1.2	0.9
Consumer_jpeg_c	1.1	1
consumer_tiff2bw	1.6	1.2
dijkstra	2.2	1.5
patricia	1.8	1.4
security blowfish	1	1.2
Susan	1.3	1.3

The above table shows execution speed up between advanced combined elimination algorithm and branch and bound algorithm. From the above table we can conclude that advanced combined elimination gives best execution speed up for most of the benchmark applications when compared to branch and bound algorithm.

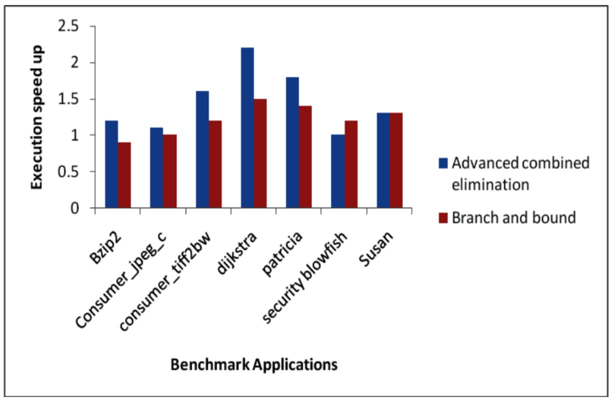

Fig. 1 Execution speed up between Advanced combined elimination and branch & bound

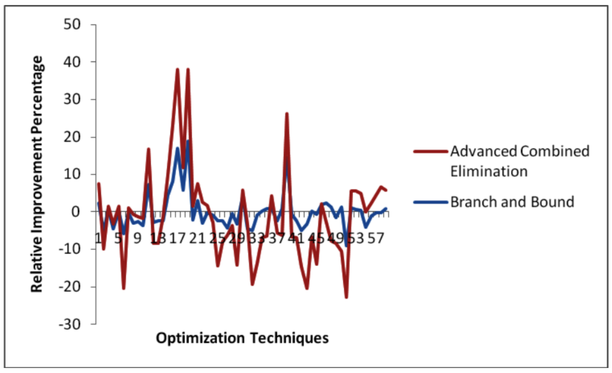

Fig. 2 Relative improvement percentage for Dijkstra

Figure 2 shows relative improvement percentage for a Dijkstra's benchmark applications. Relative improvement percentage is measured using equation (2). The above figure plotted only after the optimal set is found for a given benchmark applications after much iteration. The technique which gives an negative impact eliminated automatically during iteration.The techniques which gives positive impact included in an optimal set. Similarly relative improvement percentages for other benchmark applications are measured in this fashion.

V. CONCLUSION AND FUTURE WORK

With the help of an effective orchestration algorithms we can find best set of optimal techniques for a given benchmark applications. Advanced combined elimination which gives better speed up when compared to branch and bound strategy.In future we can design effective framework for selecing optimal set by consider other compilers such as ROSE [9], LLVM[7] and open path [8] compiler. In future we can design a generalized framework independent of architecture by considering larger set of benchmark applications.

REFERENCES

[1] GCC online documentation http://gcc.gnu.org/onlinedocs/

[2] Z. Pan and R. Eigenmann (2006), 'Fast and effective orchestration of compiler optimizations for automatic performance tuning', In proc.of the int.symp.on code generation and optimization, pp. 319-332.

[3] Z. Pan and R. Eigenmann (2004), 'Compiler optimization orchestration for peak performance', In proc. of the int.symp. on code generation and optimization, pp. 319-332.

[4] Mathew R. Guthaus, Jeffry S. Ringberg et al.(2001), 'Mibench:A free commercially representative embedded benchmark suite', Workload characterization, 2001. WWC-4.IEEE int.workshop on, pp.3-14.

[5] William Von Hagen(2006), 'The definitive guide to GCC', Apress publications, second edition, pp.101-117.

[6] Brian J. Gough (2005), 'An introduction to GCC for gcc and g++', Revised edition, pp 45-53.

[7] LLVM: the low level virtual machine compiler infrastructure. http://llvm.org.

[8] Open 64: an open source optimizing compiler. www.open64.net.

[9] ROSE:www.rosecompiler.org.

J.Andrews received the B.E degree in Computer Science & Engineering from Dr.sivanthi Aditanar college of Engineering, Manonmanium Sundaranar University Tirunelveli, India in 1999 and M.E degree in Computer Science & Engineering from Sathyabama University, Chennai, India in 2006. He is currently doing research in the area of Compiler Optimization in Computer Science & Engineering at Sathyabama University, Chennai, India.

He works currently as an Assistant Professor for the Department of Information Technology at SRR Engineering College, Chennai and he has more than 10 Years of teaching experience. He has participated and presented many Research Papers in International and National Conferences. His area of interests includes Compiler Design, Theory of Computation.