ISSN No. 0976-5697

Volume 5, No. 7, September-October 2014

International Journal of Advanced Research in Computer Science

REVIEW ARTICLE

Available Online at www.ijarcs.info

System and Method for Detecting Potential Places for Memoization in Recursive Functions of a Computer Programming Code

Narasimha Sarma Sridhara Chimmapudi
Digital Engg.-RMI&O
HCL Technologies Limited
1st Phase, Electronic City, Bangalore - India

Abstract: A system and method for detecting potential places for memoization in recursive functions of a computer programming code, includes a software tool to detect the suitable places for memoization in a source code of a program/software program. The tool automatically detects the potential places for implementing memoization in a program, the need of memoization during run time, and suggests the code to be inserted for memoization. The recursive functions of potential memoization are memoized either automatically or manually to facilitate efficient execution of the program. By implementing memoization, the complexity of algorithm would be reduced from exponential order to polynomial order and improves the speed of execution.

Keywords: Automatic detection, memoization, recursive function, software tool, runtime analysis

i. INTRODUCTION

In computer programming, dynamic programming problems are complex in nature. The complexity occurs because most of the programmers do not attempt to solve dynamic problems with sound dynamic programming techniques. Very few expert programmers can understand and solve these problems in the structured way that is recommended by dynamic programming. The major problem is most of the programmers rely on recursion and forget the memoization to solve these problems. Memoization is one of the optimization techniques used for accelerating the speed of the program execution.

Currently all commercially available run time code analysis tools like Rational Purify, Bounds Checker, Para-soft code-wizard etc. are not detecting memoization. Resultantly we need to perform memoization manually on huge source code, which is cumbersome and error prone. Code analysis tools like purify, execute typical workflows to get a detailed report on the number of times each function is run with the same inputs. If we solve these problems with recursion and without using memoization the respective algorithm's complexity will be in the exponential order. These problems would generally occur in dynamic programming and sometimes in general programming too. To avoid this complexity, the method of detecting and including memoization in recursive functions using suitable tool has been developed. If we apply memoization, the respective algorithm's complexity would be reduced from exponential order to either polynomial order or linear order in the ideal case.

We have developed a software tool for analyzing a computer programming code for detecting potential places for memoization; the analysis gives detailed report on the potential places for memoization with line numbers in a huge programming source code. After the analysis, automemoization can be performed, thereby increasing the speed of a software performance.

ii. METHOD AND MECHANISM

In this Paper, the overall method and mechanism of an analysis of a software programming code followed by automemoization is illustrated. The method of code analysis and automemoization comprising following steps;

A. Program Input

The program input is fed to the code analysis tool to detect potential places for memoization in recursive function to facilitate efficient execution of the program [1].

B. Recursive Function

The tool analyzes the time taken for multiple iterations of the direct and indirect recursive function i.e., the function with loop, with same inputs during runtime [2, 3].

C. Detecting Potential Places

Based on the above analysis of time taken for multiple iterations of recursive function, the tool will point out the list of potential places for memoization in a given program.

D. Storing of Parameter Values

While calculating, the tool will store the values of parameters in addition to the function name, number and type of parameters of a function in a temporary storage medium (cache memory) [4].

E. Injection of Memoization Code

After the analysis of recursive function, the tool will illustrate where the memoization code can be injected. The injection of memoization code in recursive function can be done either automatically or manually by the programmer [5].

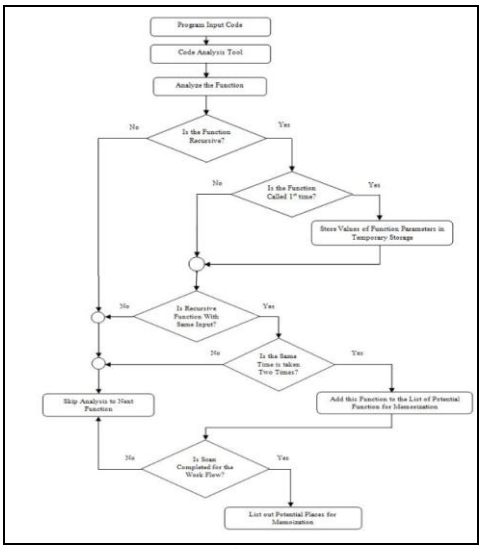

Figure 1: Mechanism of Code Analysis Tool

Figure 1 describes the method and mechanism of an analysis of a software programming code and automemoization.

iii. DESCRIPTION

System and over all method for analyzing a software source code followed by automemoization is illustrated in figure 1. In this System, a developed software tool is used for analysis a source code of a software /computer program for memoization.

A software source code is fed to the code analysis tool for detecting and implementing memoization at potential places in recursive functions to facilitate efficient execution of the software/program [3].

The code analysis tool analyzes the time taken for multiple iterations of the direct and indirect recursive function (function with loop) with same inputs during run time [4]. Then based on above analysis the tool will point out the potential places for memoization in a given program.

A recursive function having more than one recursive call may happen in two ways: a) Explicitly calling the function more than once, or b) calling this recursive function in a loop, where the memoization has not been implemented explicitly. So, the tool to detect the possibility of implementing memoization provides a scaffold/model implementation for immediate performance benefits.

While calculating, the tool will store the values of parameters in addition to function name, number and type of parameters of a function in a temporary storage medium (cache). After the analysis of recursive function, the tool will illustrate where the memoization code can be injected to facilitate efficient execution of a program.

The injection of memoization code in recursive function can be done manually by the programmer. Thus the complexity of algorithm will be reduced from exponential to either polynomial order or linear order in ideal cases and improves the performance by accelerating the speed of program execution. It is even possible to add intelligent code automatically to support memoization that enables improving the performance of the algorithm from exponential to polynomial [1].

The tool has to be applied on Static and global functions, but not on member functions of a class. It can be applied also on library functions as well. Unlike other run time analysis tools, this tool will store the parameter values in addition with function name, number of parameters and type of parameters while calculating repeated sub-solutions.

iv. AN EXAMPLE

A sample C++ program is provided below to illustrate the effect of implementing the Programming/software code analysis tool.

```
TestMemorizationSample.cpp
#include <iostream>
using namespace std;
/*
Assumption: f(n) = 1 if n <= 2
= f(n-1) + f(n-2) if (n>2)
*/
__int64 getnthfebnum_slow(int n)
{
if (n > 2) return getnthfebnum_slow(n-1) +
getnthfebnum_slow(n-2);
return 1;
}
__int64 memo[200];
__int64 getnthfebnum_fast(int n)
{
if (memo[n]) return memo[n];
if (n>2) memo[n] = getnthfebnum_fast(n-1) +
getnthfebnum_fast(n-2);
else memo[n] = 1;
return memo[n];
}
void print(__int64 n)
{
if (n<0) { cout << '-'; print(-n); return;}
if (n==0) { cout << '0'; return; }
if (n>9) print(n/10);
cout << int(n%10);
}
void main()
{
//__int64 n1 = getnthfebnum_slow(50);
__int64 n1 = getnthfebnum_fast(50);
print(n1);
}
```

With n=50, without using memorization (getnthfebnum_slow) the resultant value would be returned in several hours, while with the same n=50, with memorization (getnthfebnum_fast) the resultant value would be returned in fraction of a second.

v. ANALYSIS RESULT

Table I. gives report of memoization after a sample run of our code analysis tool on the target programming code:
In the sample table I. below, the recursive function “Add” requires memoization, because the time taken for execution is same for two iterations, where the function “Febo” does not

require memoization. From this, the programmer implements memoization in the respective function "Add" to improve code performance. It is even possible to add intelligent code automatically to support memoization that enables improving the performance of the algorithm from exponential to polynomial.

Table I. Output of Code Analysis Tool

Sr.No	*Recursive Function Name*	*Input values*	*Iteration*	*Time taken for execution*
1	Add	10 20	1	30 micro sec
2	Add	10 20	2	30 micro sec
3	Febo	20	1	50 micro sec
4	Febo	20	2	1 micro sec

vi. CONCLUSION

The developed code analysis tool is for analyzing a computer programming /software source code for potential places for memoization during the runtime and gives the report about the potential places with line number of the programming source code, which gives a clear idea on the target programming code /software source code. So, the recursive functions can be memoized automatically or manually to facilitate efficient execution of the program. Therefore, the complexity of algorithm would be reduced from exponential order to polynomial order and improves the speed of execution. The Performance improvement in this case is in the order of 10^5 - 10^6. The underlying complexity of the algorithm would be reduced from exponential to either polynomial or Linear.

vii. ACKNOWLEDGEMENT

I would like to take this opportunity to thank my colleagues Mr. Venkatraman Rajagopalan, Mr. Inderjeet Singh and Mr. Gokulanavaneethakannan for their consistent help and support. Their valuable support and help made me to complete and publish this journal successfully.

viii. REFERENCES

[1] Eric Snow, Eric Aubanel, and Patricia Evans, "Dynamic parallelization for RNA structure comparison", May 2009, IEEE Parallel & Distributed Processing, IPDPS 2009, pp. 1–8, doi: 10.1109/IPDPS.2009.5160926.

[2] Joxan Jaffar, Andrew E. Santosa and Razvan Voicu, "Efficient memoization for dynamic programming with ad-hoc constraints", 2008, Conference paper - Association for the Advancement of Artificial Intelligence, Available from: https://www.comp.nus.edu.sg/~joxan/papers/opt.pdf

[3] James Mayfield, Marty Hall and Tim Finin, "Using Automatic Memoization as a Software Engineering Tool in Real-World AI Systems", 1995, Proceedings from 11th conference on IEEE Artificial Intelligence for Applications, doi:10.1109/CAIA.1995.378786.

[4] TobinHarris, "Flyweight Pattern", July 2013, Available from: http://c2.com/cgi/wiki?FlyweightPattern

[5] Marty Hall, and J. Paul McNamee, "Improving software performance with automatic memoization", vol. 18, issue 2, 1997, Johns Hopkins APL Technical Digest, pp. 254-260.