On the Optimal Object-Oriented Program Re-modularization

Saeed Parsa and Omid Bushehrian

Faculty of Computer Engineering, Iran University of Science and Technology
{parsa,bushehrian}@iust.ac.ir

Abstract. In this paper a new criterion for automatic re-modularization of object-oriented programs is presented. The aim of re-modularization here is to determine a distributed execution of a program over a dedicated network of computers with the shortest execution time. To achieve this, a criterion to quantitatively evaluate performance of a re-modularized program is presented as a function. This function is automatically constructed while traversing the program call flow graph once before the search for the optimal re-modularization of the program and considers both synchronous and asynchronous types for each call within the call flow graph.

1 Introduction

With the increasing popularity of using clusters and network of low cost computers in solving computationally intensive problems, there is a great demand for system and application software that can provide transparent and efficient utilization of the multiple machines in a distributed system [2][3][5]. There are a number of such application softwares including middle-wares and utility libraries which support parallel and distributed programming over a network of machines. A distributed program written using these middle-wares comprises a number of modules or distributed parts communicating by means of message passing or asynchronous method calls.

Our aim has been to develop automatic techniques to obtain maximum execution concurrency among distributed parts or modules of a program. To reach this end, the main difficulty is to determine theses distributed parts, or equivalently, the architecture of a distributed program code. The architecture of a program can be reconstructed using software reverse engineering and re-modularization techniques [1][4].

2 The Optimal Re-modularization of a Program

Each clustering of a program call graph, which is a modularization of that program, represents a subset of program method calls, named *remote-call* set, to be converted to remote asynchronous calls. For instance consider the modularized call graph of four classes in Figure 1. This modularization corresponds to *remote-call* set {c1, c2, c4}.

Y. Shi et al. (Eds.): ICCS 2007, Part I, LNCS 4487, pp. 599 – 602, 2007.

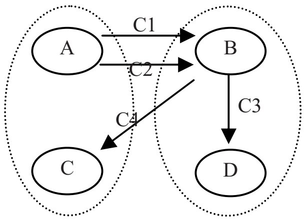

Fig. 1. Re-modularization of a program call graph

3 Performance Estimation of a Re-modularized Program

In order to evaluate a re-modularized program performance, the *remote-call* set corresponding to that re-modularization is obtained and evaluated by applying a function called *Estimated Execution Time* (*EET*). For a given *remote-call* set *r*, $EET_I(r)$ calculates a value which is an estimation of the amount of execution time of method call *I* with respect to *r*. Each *EET* formula is generated from the program call flow graph (CFG). CFG shows the flow of method calls among program classes. Each node in this graph represents a method body in an abstract way by means of a sequence of symbols. Each symbol in this sequence indicates one of these concepts: a method invocation, a synchronization point between caller and callee methods or an ordinary program instruction which are denoted by I_i, S_i and W_i respectively. Symbol S_i indicates the first program location which is data dependent to a method invocation I_i in the CFG node sequence. Symbol W_i represents any collection of ordinary program statements with estimated execution time i. Below in Figure 2 is a sample Java code and its corresponding CFG. EET function for a program is generated automatically by traversing the program CFG. Since each method invocation I_i in the program CFG may be executed either synchronously or asynchronously, depending on the specified modularization of the program classes, the EET function includes time estimation for both synchronous and asynchronous execution types for each invocation I_i. For instance the EET function for CFG in Figure 2 is generated as follows:

```
Calss A{
……  public static void main(string[] arg){
        int r1,r2;
        B b=new B(); C c=new C();
        r1= b.m(); // I1
        r2= c.n (); // I2
        use(r2);   //S2
        use(r1);   //S1
    }
private void use(int r){…}
 }//Class
```

```
Calss B{
…..
public int m(){
 //some computations
}}//Class
------------------
Calss C{
…..
public int n(){
 //some computations
}}//Class
```

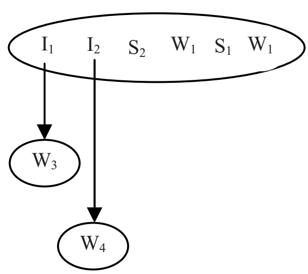

Fig. 2. A sample program including three classes and its CFG

$$EET_{main}(r)= a_1*EET_{I1}(r) + a_2*EET_{I2}(r) + (1-a_2)*T(S_2) + W_1 + (1-a_1)*T(S_1) + W_1 \quad EET_{I1}(r)=W_3 , EET_{I2}(r)=W_4 , \tag{1}$$

In this relation, depending on the execution type of invocations I_1 and I_2, asynchronous or synchronous, coefficients a_1 and a_2 are set to 0 or 1 respectively. S_1 and S_2 are synchronization points of calls I_1 and I_2 respectively and $T(S_i)$ indicates the amount of time that should be elapsed at synchronization point S_i until invocation I_i is completed.

The general form of an EET relation for a program is as follows:

$$EET_m(r)=\sum w_i + \sum a_i* EET_{Ii} (r) + \sum (1-a_i)*T(S_i) \tag{2}$$

In the above formula coefficients a_i are determined by *remote-call* set *r* as follows:

$$a_i=\begin{cases} 1 & : I_i \notin r \\ 0 & : I_i \in r \end{cases}$$

As described above, $T(S_i)$ is the amount of time that should be elapsed at synchronization point S_i until invocation I_i is completed. $T(S_i)$ is calculated by the following relation:

$$T(S_i)= \max((EET_{Ii} (r) + O_i) - t_i ,0) \tag{3}$$

Where, t_i is estimated execution time of the program fragment between symbols I_i and S_i. Since each asynchronous method invocation I_i imposes a communication overhead on the overall program execution time, this overhead which is denoted by O_i , is added to the estimated execution time of I_i. Since it is assumed that CFG is cycle free, $EET_m(r)$ can be solved by recursively replacing EET terms until $EET_m(r)$ contains only a_i coefficients, W_i terms, O_i terms and *max* operators.

4 Conclusions

The main difficulty in obtaining a distributed execution of a program with minimum execution time is to find the smallest set of program invocations to be converted to remote asynchronous invocations. Program re-modularization can be applied as an approach to reach this end. Program re-modularization is used to reconstruct program architecture with respect to one ore more quality constraints such as performance or maintainability. In this paper a new criterion for performance driven re-modularization of a program has been proposed. This criterion is used to quantitatively estimate the performance of a re-modularized program with a function which is generated automatically from the program call flow graph (CFG). This function includes time estimations for both asynchronous and synchronous execution types of each method in the program call flow graph.

References

1. Berndt Bellay, Harald Gall, "Reverse Engineering to Recover and Describe a Systems Architecture", Development and Evolution of Software Architectures for Product Families, Lecture Notes in Computer Science(1998), Volume 1429 .
2. Bushehrian Omid, Parsa Saeed, "Formal Description of a Runtime Infrastructure for Automatic Distribution of Programs", The 21th International Symposium on Computer and Information Sciences, Lecture Notes in Computer Science(2006), Vol. 4263.
3. Jameela Al-Jaroodi, Nader Mahamad, Hong Jiang, David Swanson, "JOPI: a Java Object Passing Interface", Concurrency Computat. : Pract. Exper. (2005); 17:775–795
4. Parsa S. , Bushehrian O., "The Design and Implementation of a Tool for Automatic Software Modularization", Journal of Supercomputing, Volume 32, Issue 1, April (2005).
5. Parsa Saeed, Khalilpour Vahid, "Automatic Distribution of Sequential Code Using JavaSymphony Middleware", SOFSEM06, Lecture Notes in Computer Science(2006), Vol. 3831,.