Journal of Zhejiang University-SCIENCE C (Computers & Electronics)
ISSN 1869-1951 (Print); ISSN 1869-196X (Online)
www.zju.edu.cn/jzus; www.springerlink.com
E-mail: jzus@zju.edu.cn

Automatic actor-based program partitioning

Omid BUSHEHRIAN

(*Department of Information Technology, Shiraz University of Technology, Shiraz 71555-313, Iran*)

E-mail: bushehrian@sutech.ac.ir

Received Feb. 16, 2009; Revision accepted June 8, 2009; Crosschecked Sept. 29, 2009

Abstract: Software reverse engineering techniques are applied most often to reconstruct the architecture of a program with respect to quality constraints, or non-functional requirements such as maintainability or reusability. In this paper, AOPR, a novel actor-oriented program reverse engineering approach, is proposed to reconstruct an object-oriented program architecture based on a high performance model such as an actor model. Reconstructing the program architecture based on this model results in the concurrent execution of the program invocations and consequently increases the overall performance of the program provided enough processors are available. The proposed reverse engineering approach applies a hill climbing clustering algorithm to find actors.

Key words: Actor model, Software reverse engineering, Performance evaluation
doi:10.1631/jzus.C0910096 **Document code:** A **CLC number:** TP31

1 Introduction

One of the most important aspects of the quality of software is its performance. Many legacy programs can be reengineered to optimize their performance. However, existing reengineering techniques are dedicated to reconstructing the architecture of a program with respect to quality constraints such as maintainability or reusability (Mitchell and Spiros, 1999; Tahvildari *et al*., 2003; Parsa and Bushehrian, 2005) and they have not been used to assess the architecture of a program from a performance viewpoint. All current architectural level performance engineering techniques are focused on performance assessment in the early stages of the software development life cycle (Hyunsang *et al*., 2007; Joao *et al*., 2007; Robert and Hassan, 2007). However, the implemented software still may not meet its performance provisions and may need to be optimized to improve performance. In this paper, AOPR, a novel actor-oriented program reverse engineering approach is presented to reconstruct an object-oriented program architecture based on a high performance model such as the actor model (Agha and Thati, 2004). According to this model, actors interact asynchronously while all communications inside an actor are performed synchronously. Therefore, reconstructing the program architecture based on this model results in the concurrent execution of the program invocations and consequently increases the overall performance of the program when enough processors are available. The question is how to automatically produce actors from the program source such that the highest amount of concurrency is obtained in the execution of the actors over a cluster or multiprocessor. This research answers two basic questions: first, how can a sequential object oriented program be partitioned into concurrent parts (each called an actor) such that by deploying those parts over cluster nodes the overall program performance is increased? Second, how can program statements be reordered such that more concurrency is achieved while preserving the sequential semantics? A major difficulty is that sequential programmers are used to apply the results of any method call immediately. Therefore, there is no opportunity for concurrency when translating ordinary method calls into asynchronous calls among actors. To resolve this difficulty, for the first time, we have applied the ideas of instruction scheduling to increase the distance between each inter-actor call instruction and the very first instructions applying the call results (Maani and Parsa, 2007). Our algorithm attempts to insert as many instructions as possible between an asynchronous remote invocation and its first data-dependent statement, considering the data dependencies between the statements. In addition to instruction scheduling, suitable partitioning of a program can result in higher concurrency. The objective of most partitioning techniques used in reverse engineering of distributed programs has been to minimize the communication cost and enhance performance in terms of speedup (Gourhant *et al.*, 1992; Deb *et al.*, 2006). However, as shown in this paper, in addition to minimizing the communication cost, the amount of concurrency in execution of the actors needs to be considered as an objective of the partitioning. To this end, in this paper, a new performance evaluation function is presented which is used to evaluate each partitioning of the program classes. The partitioning, *p*, that minimizes this function is selected and each partition in *p* is assumed as an actor.

The main contribution of this paper to the existing literature is two-fold. First, it provides a new performance evaluation function, extracted automatically from a program call flow graph, to evaluate the performance of any partitioning of the program. This function is applied as the objective function in a hill climbing partitioning algorithm to find a near-optimal partitioning of the program from the performance viewpoint. Second, it presents a new optimization technique that uses instruction reordering to enhance the concurrency in the execution of asynchronous method calls among actors by increasing the distance between each call instruction and the very first instruction applying the call results.

2 Related studies

Current research in software performance engineering (SPE) is dedicated to estimating the performance of software in the early stages of the development process because of the need for quality of service (QoS). To achieve this goal, several studies have been carried out to transform software architectural models into analyzable formal models. Examples include deriving queuing network models from unified modeling language (UML) diagrams (Hyunsang *et al.*, 2007) or translating some of the UML diagrams to Perti Nets (Joao *et al.*, 2007; Robert and Hassan, 2007). Andolfi *et al.* (2000) proposed constructing and analyzing two kinds of performance models: a software execution model and a system execution model. The former represents the software execution behavior and is modeled by execution graphs, and the latter is based on queuing network models, which represent the computer system platform, including hardware and software components. The software and system execution models are applied to assess the performance of the intended software architecture. Some related research has also been carried out in the area of performance optimization of existing programs. In a mixed dynamic and programmer-driven approach to optimize the performance of large-scale object-oriented distributed systems, Gourhant *et al.* (1992) proposed an object-partitioning method dynamically invoked at runtime to collocate objects that communicate often. Here, the partitioning criterion is to gather in the same partition objects that communicate often. In a distribution strategy for program objects, Deb *et al.* (2006) presented an object graph construction algorithm and a partitioning policy for program objects based on object types. The distribution strategy is to place objects communicating most often in the same machine to minimize network traffic. However, when partitioning a program, in addition to minimizing the communication cost, the amount of concurrency in the execution of the distributed partitions needs to be maximized.

Most studies of the automatic partitioning of object-oriented programs are dedicated to developing a programming model and a partitioning or parallelizing compiler. SCOOP (Sobral and Proenca, 1999) is a parallel programming model based on object-oriented paradigm. The SCOOP methodology allows the programmer to specify the number of slaves and masters (parallel objects) and a number of parallel tasks (method invocations). The SCOOP runtime system packs some of the masters and slaves and aggregates method invocations to reduce the impact of the overhead caused by excess parallelism and communication. Mentat (Grimshaw, 1993) uses a partitioning compiler and a runtime support in conjunction with the object-oriented paradigm to produce an easy-to-use high-performance system that facilitates hierarchies of parallelism. Mentat accomplishes these objectives through two primary components, Mentat programming language and the underlying runtime system. JavaParty (Philippsen and Zenger, 1997) is a minimal extension to Java that facilitates distributed programming for cluster computers. A source code transformation automatically generates a distributed Java program based on remote method invocation (RMI). Orca (Bal and Kaashoek, 1993) is a language for parallel programming of distributed systems based on the shared data-object model. All these models perform program partitioning by introducing new keywords into the programming languages (such as Java). Then, the programmer specifies the partitioning he/she thinks is best using these keywords. In contrast, AOPR determines the best partitioning by applying a heuristic search algorithm.

3 Static analysis of the program

As described earlier, all actors interact through asynchronous calls while all local invocations within an actor are synchronous. There are several difficulties when transforming a program into a set of communication actors. First, the synchronization point for each method call within the program needs to be determined. The synchronization point of a method call is the very first data-dependent statement to that call. Second, the optimal partitioning of the program needs to be determined. Third, the inter-actor method calls need to be transformed into asynchronous calls. These problems are addressed in the following sections.

3.1 Synchronization points

The synchronization point for an asynchronous call is defined as the point in the program where the first data-dependent instruction on the call statement appears. To determine the first statement which is data-dependent on a call statement, we have applied the data access summaries approach (Chan and Abdelrahman, 2004). In this approach, for each method *m*, symbolic access paths are defined. Symbolic access paths for a method represent access to three types of objects: global objects (which may be used by static calls), objects which are formal parameters of *m*, and the object 'this' within the method. The symbolic access paths of a method are applied to generate the data access summary for the method. The data access summary for a method *m* is a set of pairs ($n, <type>) where the anchor $n denotes the *n*th formal parameter of *m*, <type> indicates the type of access, which may be 'read' or 'write', and $0 stands for the object 'this'. For instance, consider the following class declaration:

```
class A {int a;
   Public void m(B, b, int t)
   {a=b.get(); b.set(t);}}
```

In the above example, the method *m* writes to the member variable *a*. This is represented by the pair ($0, write) in the data access summary for the method *m*. Before the access type for the first formal parameter, *b*, can be determined, it is necessary to work out the data access summary for the method set() of the class *B*. For instance, if the data access summary for the method set() includes the pair ($0, write), then the statement *b*.set(*t*) writes to the object *b*. This is represented by the pair ($1, write) in the data access summary for the method *m*. After the data access summary for each method within a program is formed, it is easy to find the synchronization point for each method call because the data access summary for each invoked method indicates whether the parameters passed to the method are written by it. The synchronization point for a method call, *I*, is the very first position within the caller where any value written to by *I* is used. For example, consider the following piece of code:

```
a.m(b1, t1); //access summaries for m: ($0, write), ($1, write)
if (condition) {
…
b1.n(); //Synchronization point of a.m(b1, t1)
…}
```

According to the access summaries for *m*, the invocation *a*.*m*(*b*1, *t*1) writes to *a* and *b*1. The first statement which uses *a* or b1 is the synchronization point for the method call *a*.*m*(*b*1, *t*1). Obviously, data access summaries for any static method call do not include the pair ($0, <type>).

Moreover, a method call I_i may write to a global resource such as a static variable or an external file. The data access summaries for I_i can be extended to include some pairs (global_object_name, <type>) to address these kinds of dependencies. These additional pairs are used to determine the data-dependent statements on I_i and its synchronization point.

3.2 Transformation of local into asynchronous invocations

After determining the partitioning of a program, all inter-actor invocations are transformed into asynchronous calls. To translate an ordinary method call *a.m*(p1, p2, …, p*n*), in an actor A_i, into a corresponding asynchronous call, two approaches have been implemented in AOPR. In the first approach, the method call is transformed into a remote asynchronous method call supported by JavaSymphony middleware (Fahringer and Jugravu, 2002). This approach is used when actors are deployed over a cluster for execution and can be applied to every method call including static method calls. Another approach is to replace a method call with a statement that invokes that call inside a separate kernel level thread. To do this, a 'launcher' class is generated automatically for each method call that is detected as an inter-actor call. This is performed for both object and static method calls. For example, consider the inter-actor method calls *a.m*(p1, p2) and *A.n*(p1, p2), in which *a* is an object reference of type *A*, and p1 and p2 are object references of types P1 and P2 respectively, and *n* is a static method inside class *A*. The first call statement is replaced with the following statements:

```
Semaphore sem1=new Semaphore(0);
l1=new launcher1(a, p1, p2, sem1);
l1.start();
```

In the above code, launcher1 is the launcher class corresponding to method call *a.m*(p1, p2). This class extends Java thread and starts this method call in the context of a separated thread. sem1 is a semaphore object which enables the parent thread to wait for results of the method call *a.m*(p1, p2) at its synchronization point. The constructor of the launcher class corresponding to the static method call *A.n*(p1, p2), accepts three parameters:

```
Semaphore sem2=new Semaphore(0);
l2=new launcher1(p1, p2, sem2);
l2.start();
```

If a global object is accessed by more than one thread, for instance, by means of a static accessor method, that accessor method is declared to be synchronized.

4 AOPR: actor-oriented program reverse engineering

The actor model unifies the notion of objects with concurrency (Agha and Thati, 2004). Each actor operates asynchronously with other actors by sending asynchronous messages. In this paper, AOPR is applied to transform an object-oriented program into a set of actors communicating by means of asynchronous method calls. Since some of the blocking invocations among objects in the program are transformed into asynchronous calls among actors, the overall performance of the program may be increased. The idea is to partition the program classes and consider each partition as an actor. Therefore, each actor, which is a partition of the program classes, contains one or more classes communicating synchronously (intra calls), while invocations to the objects outside this actor (inter calls) must be transformed to asynchronous calls. There are different ways to partition program classes to form actors. However, we should search for the partitioning with the highest possible concurrency resulting from asynchronous calls. To achieve this, the amount of concurrency corresponding to each partitioning of the program classes needs to be estimated. Therefore, in this paper a function for estimating the amount of concurrency among the extracted actors is proposed. This function is then used by a heuristic hill climbing search algorithm to evaluate each possible partitioning in the search space and to find the partitioning of the program classes with the highest amount of concurrency. Since the problem of partitioning is an NP-complete problem, heuristic search algorithms such as hill climbing are usually used to find the near-optimal solution. The main steps in AOPR can be summarized as follows:

1. Extract a call graph from the program source code.

2. Extract the performance evaluation function (PEF) by analyzing the program source code.

3. Search for a partitioned call graph that minimizes the amount of PEF. This is achieved by using a heuristic search algorithm such as hill climbing that

uses PEF as its objective function. These steps are depicted in Fig. 1.

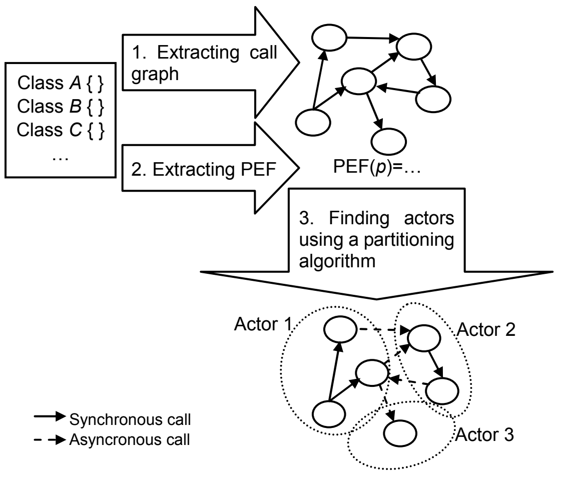


Fig. 1 AOPR steps: the PEF function evaluates each possible partitioning *p* of the program and finds the partitioning with the best performance

4.1 Call graph partitioning

The first step in AOPR, is analyzing the program to extract its call graph. The call graph is a model of program source code representing the classes and invocations among them. For each pair of classes such as (A, B) in the program call graph, it should be determined whether A and B share the same partition or should be placed in different partitions. This decision is made by the partitioning algorithm. This algorithm evaluates each possible partitioning of the program call graph using a PEF. Eventually, the partitioning which minimizes this function is selected. Searching among all possible partitioning of a call graph is an NP-complete problem. Therefore in this paper a hill climbing algorithm is applied to find the near-optimal solution using PEF as its search objective function. The partitioning selected by the hill climbing algorithm represents the internal structure and external communications of extracted actors. All the classes inside an actor communicate via synchronous method calls while all the invocations among classes belonging to different actors are transformed into asynchronous calls.

4.2 Hill climbing objective function

The PEF is built automatically while traversing a program call flow graph. A call flow graph represents the flow of method calls among program classes. In this graph, nodes represent method bodies and edges represent invocation among methods. The PEF is built by considering the estimated execution time of all instructions within the program code, including the invocations. Because the type of invocation (synchronous or asynchronous) affects the overall execution time of the program and is not determined until the program is partitioned, the PEF is built as a general function including both invocation types for each method call. The hill climbing search algorithm then applies the PEF when evaluating the performance for each partitioning of the program call graph and selects the partitioning which minimizes the PEF. For instance, consider a method invocation I_1 that performs another invocation I_2 during its execution. The estimated execution time of I_1, denoted by T_{I1}, for both the synchronous and asynchronous types of I_2, is shown by

$$\begin{cases} T_{I1}^{\text{in}} = T_0 + T_{I2}, \\ T_{I1}^{\text{out}} = T_0 + s + S_2, \end{cases} \tag{1}$$

$$S_2 = \max(T_{I2} + 2O - d_2,\ 0). \tag{2}$$

In the above equations, T_{I1} and T_{I2} are the estimated execution time of I_1 and I_2 respectively. T_{I1}^{in} denotes the estimated execution time of I_1 when I_2 is an invocation to an object inside the current partition (the caller and called objects share the same partition), and therefore I_2 is performed synchronously. T_{I1}^{out} denotes the estimated execution time of I_1 when I_2 is an invocation to an object outside the current partition (the caller and called objects reside in different partitions) and therefore I_2 is performed asynchronously. Assuming that the target of the invocation I_1 is method m, T_0 is the total execution time of all the instructions excluding I_2 within m. When an invocation such as I_2 is performed asynchronously, the caller should wait for the results of I_2 at some synchronization point during its execution. In Eq. (2), S_2 denotes the amount of required time at the synchronization point of I_2 to receive the results of method call I_2. Parameter O in Eq. (2) denotes the incurred communication overhead between the caller and the receiver when I_2 is performed asynchronously. Parameter s denotes the amount of time required for initiating the asynchronous call.

Eqs. (1) and (2) can be combined as a single equation as follows:

$$\begin{cases} T_{I1} = T_0 + a_1 T_{I2} + (1-a_1)(s+S_2), \\ S_2 = \max(T_{I1} + 2O - d_2, 0). \end{cases} \tag{3}$$

In Eq. (3), a_1 is a Boolean variable whose value is either 0 or 1 depending on whether I_2 is invoked asynchronously or synchronously. There may be several invocations, I_i, within method m and each invocation itself may include other invocations. Depending on the partitioning p of the program call graph, each invocation can be either synchronous or asynchronous. Therefore, Eq. (3) for estimating the execution time of I_1 can be generalized as follows:

$$T_I(p) = T_0 + \sum a_i T_{Ii}(p) + \sum (1-a_i)(s+S_i), \tag{4}$$

$$S_i = \max((T_{Ii}(p) + 2O_i) - d_i,\ 0). \tag{5}$$

In the above equation, S_i denotes the time elapsed to wait for the results of the invocation I_i, d_i denotes the estimated execution time of the program statements located between each call statement, I_i, and the first locations where the results of the call are required (the synchronization point of I_i), O_i denotes the communication overhead for sending the parameters and receiving the return values of I_i. The values of the parameters a_i in Eq. (4) are determined considering the partitioning p. If within partitioning p, I_i is an invocation between objects sharing the same partition, the value of a_i is set to 1; otherwise, a_i will be 0. Finally, $T_I(p)$ denotes the estimated execution time of invocation I considering the partitioning p, and $T_{Ii}(p)$ is the estimated execution time for invocation I_i called inside I.

The PEF(p) function that returns the estimated execution time for partitioning p of a program is built by applying Eqs. (4) and (5) recursively starting from the main() method. Assuming that the program call flow graph is cycle-free, PEF(p) can always be computed recursively. However, there may be cycles in the call flow graph, resulting from direct or indirect recursive calls. Assuming that I_i is an invocation to a method in the cycle (and itself is not in the cycle) and the estimated number of recursions is n_i then the PEF$_{Ii}$(p) should be multiplied by n_i and the back edge of the recursion should be removed from the call flow graph. In this case, Eq. (4) will be modified as

$$\mathrm{PEF}_I(p) = T_0 + \sum a_i n_i m_i \mathrm{PEF}_{Ii}(p) + \sum (1-a_i) k_i (s+S_i). \tag{6}$$

An invocation I_i or a synchronization point S_i may be located within a loop statement. Therefore, to consider the impact of loop iterations on the time estimation, coefficients m_i and k_i have been added to Eq. (6).

The time estimation in AOPR is pessimistic. It means that the elapsed time between an invocation I_i and its (first) synchronization point S_i, which is denoted by d_i, is calculated for the shortest possible path in the program between I_i and S_i. The reason is obvious: to decide whether I_i should be asynchronous, AOPR should simply compare d_i with the estimated time of I_i plus the amount of overhead for an asynchronous invocation (Eqs. (1)–(6)). Therefore, if we overestimate d_i, the method call I_i may be detected wrongly as an inter-actor call by the partitioning algorithm, badly affecting overall performance. Therefore, for if-else statements the branch with the shorter estimated execution time is considered as

```
I_i
If (condition){
//estimated execution time of this part is d_1
}
Else {//estimated execution part of this part is d_2
}
S_i
...
```

In the above code, d_i=min(d_1, d_2). Fig. 2a shows an example of a program code. The code includes three invocations, b.m1(), c.m2() and d.m3() which, depending on the partitioning, may be either asynchronous or synchronous. The PEF(p) for this program starting from the main() method is as follows:

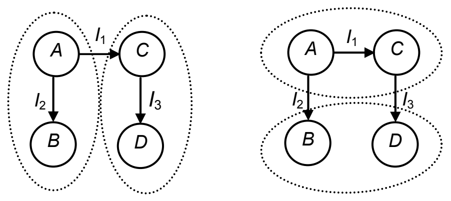
$$\begin{aligned} \mathrm{PEF}(p) &= a_1\mathrm{PEF}_{I1}(p) + a_2\mathrm{PEF}_{I2}(p) + (1-a_2)(s+S_2) + T_1 \\ &\quad + (1-a_1)(s+S_1) + T_2, \\ \mathrm{PEF}_{I1}(p) &= T_3, \\ \mathrm{PEF}_{I2}(p) &= a_3\mathrm{PEF}_{I3}(p) + (1-a_3)(s+S_3), \\ \mathrm{PEF}_{I3}(p) &= T_4, \\ S_1 &= \max((\mathrm{PEF}_{I1}(p) + 2O_1) - (a_2\mathrm{PEF}_{I2}(p) \\ &\quad + (1-a_2)(s+S_2) + T_1),\ 0), \\ S_2 &= \max((\mathrm{PEF}_{I2}(p) + 2O_2) - 0,\ 0) = \mathrm{PEF}_{I2}(p) + 2O_2, \\ S_3 &= \max((\mathrm{PEF}_{I3}(p) + 2O_3) - 0,\ 0) = \mathrm{PEF}_{I3}(p) + 2O_3. \end{aligned}$$

The values of PEF for two different partitionings of the program are presented in Fig. 2b and 2c.

```
Class A {
public static void main (string []
arg) {
      int r1, r2;
      B b=new B(); C c=new C();
      r1=b.m1();                  //I1
      r2=c.m2();                  //I2
      if (r2==−1) {…}             //S2
      //some computations: T1
      If (r1>r2) {…}              //S1
      //some computations: T2
}} //Class
```

```
Class B {
…
public int m1() {
//some computations: T3
}} //Class
Class C {
public int m2() {
   D d=new D();
   r3=d.m3();          //I3
   print(r3);
}} //Class
Class D {
public int m3() {
   //some computations: T4
}} //Class
```

(a)



(b) (c)

Fig. 2 A program code and two partitionings of this program
(a) An example of a program code; (b) p_1. Asynchronous: I_1. a_1=0, a_2=1, a_3=1, PEF(p_1)=3T+max(O−T, 0); (c) p_2. Asynchronous: I_2, I_3. a_1=1, a_2=0, a_3=0. PEF(p_2)=4T+2O. It is assumed that T_1=T_2=T_3=T and 2O_1=2O_2=2O_3=O

As discussed earlier, each partitioning of a program results in a set of actors communicating via asynchronous calls. Obviously, from a performance perspective, the best partitioning p_{optimal} is the one that minimizes the PEF function. A graph partitioning algorithm is used to search for the optimal partitioning. The PEF function generated for the program is then used as the partitioning objective function. Hill Climbing algorithms are widely used to solve graph partitioning problems (Mitchell, 2002). Therefore, the PEF function can be used as the objective function of the Hill Climbing partitioning algorithm which should be minimized. The Hill Climbing partitioning algorithm is discussed in the subsequent sections.

4.3 Automatic generation of PEF functions

The PEF function is built while traversing the program call flow graph. A call flow graph represents the flow of method calls among the program classes. Each node within this graph represents a method body and each edge corresponds to an invocation among methods. In the case of polymorphic calls, the caller is connected to all possible receivers. A pseudo code describing the algorithm for generating the PEF function is shown as follows:

Algorithm 1 GeneratePEF

```
Input: A Java program code.
Output: PEF function.
Begin
  Build the program call flow graph (CFG);
  Estimate/Input the number of iterations, $n$, of cycles in the CFG;
  Annotate all the edges within the cycle with $n$;
  Remove the cycle back-edge;
  For each node with no outgoing edges in the CFG
    Tag the node as READY;
  While there is a node, $N$, within the CFG, which is tagged
    READY do
    Let $m$ be the corresponding method name within the program;
    Estimate the execution time, $t_d$, of $m$ instructions;
    Set PEF$_N$=toString($t_d$);
  For each invocation $I_i$ in $m$ do
    Let cycles$_i$ be the label of the CFG edge corresponding to $I_i$;
    Locate the very first position $S_i$ where the call results are
      required;
    Let Iter$_i$ be the number of loop iterations enclosing $I_i$;
    Let PEF$_{Ii}$ be the PEF function for target node of invocation $I_i$
      in CFG;
    Let coef$_i$ be toString(Iter$_i$)+‘*’+toString(cycles$_i$);
    Set PEF$_N$=PEF$_N$+‘+’+coef$_i$+‘*$a_i$*’+PEF$_{Ii}$;
  End For
  For each synchronization point $S_i$ in $m$ do
    Set $d$=execution time of instructions between $I_i$ and $S_i$;
    Set $d_i$=toString($t$);
    For each $I_k$ between $I_i$ and $S_i$ do
      $d_i$=$d_i$+‘+’oef$_k$+‘*$a_k$*’+PEF$_{Ik}$;
    For each $S_k$ between $I_i$ and $S_i$ do
      $d_i$=$d_i$+SynchronizationTime$_k$;
    Set $T_{Si}$=’max(’+PEF$_{Ii}$+’2$T_c$-’+$d_i$+’, 0)’;
    Let Iter$_i$ be the number of loop iterations enclosing $S_i$;
    Let SynchronizationTime$_i$ be iter$_i$+‘*(1−$a_i$)*’+$T_{Si}$;
    Set PEF$_N$=PEF$_N$+‘+’+SynchronizationTime$_i$;
  End For
    Set a node as READY provided that the PEF function for all
      of its children has already been generated;
  End While
End Algorithm
```

5 Potential degree of parallelism

In the preceding sections, AOPR was described. The main idea was to search for the partitioning of program classes which results in the highest amount of concurrency among actors. Obviously, concurrency is achieved by performing some of the method invocations among actors asynchronously. However, converting an ordinary method call into an asynchronous one poses two kinds of overheads on the execution: initiation and communication. Initiation is denoted by s and communication by O in Eqs. (4) and (5). Therefore, from a performance perspective, converting an ordinary call I_i (between two actors) into an asynchronous call is beneficial only when the sum of execution times of program instructions located between I_i and its synchronization point S_i, denoted by d_i in Eq. (5), is greater than $s+O$. Obviously, the larger the amount of d_i, the more concurrency between the caller and the receiver is obtained.

However, a major difficulty is that programmers usually use the results of any method call I_i immediately (when d_i=0). Therefore there will be no opportunity for concurrency when transforming method calls into asynchronous calls. To resolve this difficulty, we have applied the ideas of 'statement reordering' to enhance the potential degree of parallelism of a program by increasing the amount of d_i for each invocation. The algorithm attempts to insert as many statements as possible between an invocation and its first data-dependent statement, considering the data dependencies between the statements. Obviously, the reordering should be performed such that the original semantic of the program is preserved. To do this during the statement reordering, the data and control dependency among statements must not be violated. Data and control dependency among program statements are represented by an acyclic graph called a Task Graph (Zima and Chapman, 1991). The statement reordering is performed such that the dependencies represented by the program task graph are not violated. The statement reordering technique is applied on the program source code before performing the AOPR steps.

5.1 Statement reordering

The statement reordering algorithm is presented as a method to enhance the potential degree of parallelism of a program. In this algorithm, program statements are classified as follows:

Call, a method invocation;

Use, statement which is data dependent on a Call;

Common, an ordinary statement which is neither a Call nor a Use.

The algorithm comprises two main steps. First, the program statements are moved from the original code to the reordered code gradually. In this step, Call statements are moved to the reordered code first and Use statements are moved as late as possible (Algorithm 2). Second, the reordered code resulting from the first step is further optimized by reducing the time elapsed to wait for the results of Call statements. This is achieved by inserting as many statements as possible between each Call and its corresponding Use.

Algorithm 2 Statement reordering algorithm

```
Algorithm Reorder (Task-Graph: DAG) OUT:
Reordered-Code: List
Begin
  If there is no Call in Task-Graph which is not moved
    While there is a Common-statement in Task-Graph
      which is not moved
    Select a Common-statement whose predecessors in
      Task-Graph are already moved;
    Append this statement to Reordered-Code;
    Label this instruction as moved;
    End While
    While there is a Use in Task-Graph which is not moved;
    Select a Use whose predecessors in Task-Graph are al-
      ready moved;
    Append this instruction to Reordered-Code;
    Label this instruction as moved;
    End While
  Else
    Find a Call C with the longest execution time in Task-
      Graph which is not moved;
    Let New-Task-Graph be a Sub-graph of Task-Graph, in-
      cluding Predecessors of C;
    Reorder (New-Task-Graph);
    Append C to Reordered-Code;
    Label C as moved;
    Reorder (Task-Graph);
  End If
End
```

In the first step, Call statements of longer execution time are moved to the reordered code first because the longer the execution time of a Call,the more statements should be inserted between that Call and the corresponding Use. Obviously, before a statement is moved into the reordered code, all of its parent statements in the program task graph should be moved into the reordered code. In the second step, the reordered code resulting from the first step is further optimized. To achieve this, the time required to wait for the results of Call statements is minimized. This is achieved by pushing down Use statements with positive wait time until their wait time reaches zero. Here, Use statements with longer wait time are selected and pushed down first.

6 Implementation

We have developed a support tool that implements the AOPR steps. Within this environment, a Java source code analyzer implemented using the software composition system (COMPOST) (http://www.info.uni-karlsruhe.de/~compost/CurrentCOMPOST/index.html) library, analyzes the program source to extract the call graph and call flow graph and build the PEF function. To build the PEF function for a program, first the number of clock cycles for each statement in the program is determined, according to the Java optimized processor (JOP) microinstruction definition (Schoeberl, 2006), and saved in an extensible markup language (XML) document. JOP is an implementation of Java virtual machine in hardware. Our tool also inputs the loop bounds in the program as they are needed during the PEF generation. The generated XML document is applied by a statement reordering engine to maximize the distances between each call statement and its very first data-dependent statement. The reordered program and the XML document are input to a separate module to produce the PEF function. The resultant PEF function is applied as an objective function of a hill climbing partitioning algorithm to find a near-optimal partitioning for the program. The partitioning algorithm uses PEF as an arithmetic function. AOPR applies the algorithm presented in this paper to generate the function from the program call flow graph. The programmer then copies and pastes the generated PEF in the source of the hill climbing partitioning algorithm and recompiles it. For brevity, some implementation details have not been included here. For example, PEF is first generated as a function but to make this function more (re)useable by partitioning algorithms, the generated function is enclosed in a fabricated class.

We have used the Bunch hill climbing algorithm (Mitchell and Spiros, 1999; Mitchell, 2002) to partition the call graph. Here, each partitioning of the call graph is represented by a partitioning string. The *i*th place in string *p* holds the partition number of the *i*th node in the call graph. For instance, consider a call graph with 6 nodes (Fig. 3). Since the number of partitions in every possible partitioning of this call graph cannot exceed 6, each possible partitioning of this call graph is represented by a string of six numbers, each in the range of 1–6. Fig. 3 shows a sample partitioning of this call graph and the corresponding string. The hill climbing algorithm starts with a population of randomly generated strings $P=\{p_1, p_2, \ldots, p_k\}$. For each string p_i in the population, the algorithm finds a neighbor partitioning n_k of p_i such that PEF(n_k) is less than PEF(p_i). Each neighbor partitioning of a partitioning p_i is obtained by replacing one of the partition numbers in $\mathrm{p_i}$ with another partition number. In Fig. 3, three neighbors for the partitioning string 1, 1, 2, 2, 1, 3 are depicted. The algorithm then replaces n_k with p_i in P and iterates with new P. The algorithm stops iterating when no more replacement is possible. Afterwards, the partitioning r in P whose PEF(r) is minimum is selected.

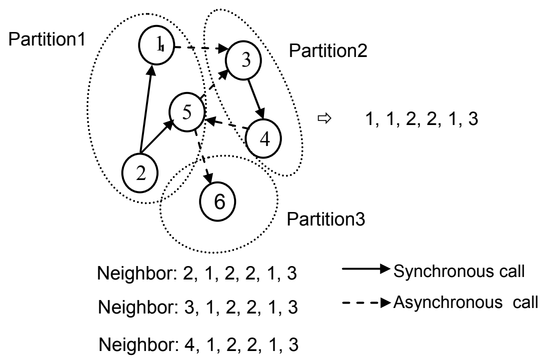


Fig. 3 A partitioning of a call graph, the corresponding string and three neighbor partitions

7 Evaluation results

A practical evaluation of the AOPR approach to optimize the performance of object-oriented programs is presented in this section. We used two Java case studies: travel sales person (TSP) and consolidated clustering (CC). The first case study evaluated the impact of applying the proposed approach on a TSP program containing 18 classes and 129 method calls. This program finds near-optimal Hamiltonian circuit in a graph, using minimum spanning trees.

The second case study measured the amount of speedup achieved by converting a program called 'consolidated clustering' (Mitchell, 2002) into a set of actors. Consolidated clustering is a graph clustering application written in Java. This program comprises 16 classes and 23 method calls. In this program, a graph is clustered several times using heuristic clustering algorithms. The results of each clustering are stored in a database for further use. This program consolidates the clustering results to obtain a clustering with a specific confidence threshold. The program is relatively slow, because it applies heuristic algorithms for clustering. The case studies were partitioned using the AOPR approach to find actors. Two different test beds were chosen to run the actors. The first test bed (TB1) was a cluster with 4 single-processor Pentium computers running JavaSymphoney (Fahringer and Jugravu, 2002) as the cluster middleware. The second test bed (TB2) was a multi-processor computer with four Pentium 2.4 GHz processors. In TB1, asynchronous invocations among actors were implemented by means of asynchronous method calls among cluster nodes supported by JavaSymphoney and in TB2 they were implemented using Java threads. The parameter passing mechanism in TB1 was implemented using the copy-restore technique. Before applying AOPR to the case studies to find actors, the values for two parameters O_i and s in function PEF (described in Eqs. (4) and (5)) were measured in the two test beds. As described earlier, O_i denotes the amount of communication overhead for passing parameters and receiving results in an asynchronous invocation and s denotes the required time for initiating an asynchronous invocation. The amount of communication cost over the Pentium cluster, associated with our underlying communication middleware, JavaSymphony, was measured as less than 100 ms. To estimate the communication cost O_i in terms of JOP clock cycles, the number of clock cycles of a sample program was divided by the measured execution time of that program. According to this estimation, the communication cost was nearly 10^7 clock cycles. The amount of communication cost in TB2 was negligible and was set to 0. The measured amounts of parameter s in both test beds were almost the same and were estimated at 10^4 clock cycles. The results of applying AOPR to these two case studies are summarized in Table 1.

Table 1 Results of applying AOPR to two case studies

Case	Number of actors		Number of asynchronous calls	
	TB1	TB2	TB1	TB2
TSP	2	4	4	12
CC	2	6	2	9

The measured speedups resulting from executing TSP and CC actors on TB1 and TB2 are shown in Fig. 4. The actors were produced using three methods: (1) applying PEF function, (2) applying MCMC function, and (3) applying a trivial method. MCMC denotes the traditional reverse engineering criterion: minimized coupling and maximized cohesion among modules. In the trivial method, all the objects were assumed actors and all the invocations in the program were transformed into asynchronous calls.

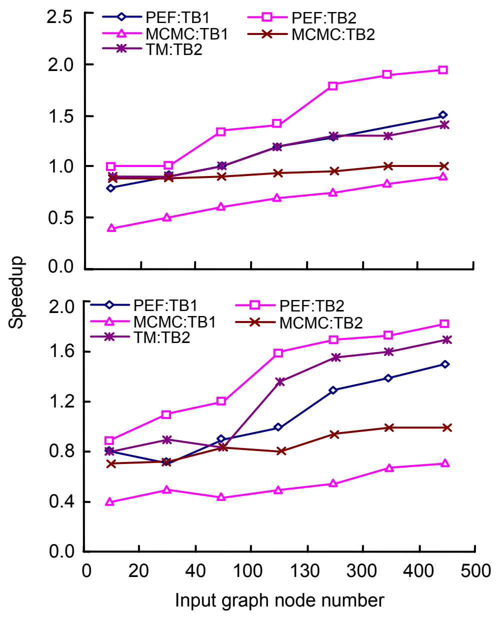


Fig. 4 Results of speedup measurement for (a) travel sales person (TSP) and (b) consolidated clustering (CC)

8 Discussion

The speedups were generally higher in TB2 (Fig. 4) as the amount of communication overhead in TB2 was negligible compared with that in TB1. The results show that the MCMC criterion is not beneficial when performance improvement is the main objective of the reverse engineering, because MCMC ignores the amount of possible concurrency when evaluating a partitioning of a program. MCMC attempts only to gather the most communicating classes in a same actor. Assume there are two classes A and B, which communicate by a large number of invocations. Applying the MCMC criterion, these two classes will be assigned to a same module. MCMC neglects the fact that the invocations between any two classes such as A and B may have the possibility of concurrent execution. Even the trivial method (TM) works better than MCMC on TB2 as all invocations including those with a potential degree of parallelism are performed asynchronously by converting them into Java Threads.

9 Conclusion

In this paper we have implemented a new actor-oriented program reverse engineering approach called AOPR, applied to reconstruct the architecture of existing software based on the actor model. In AOPR, first an architectural level performance assessment function called PEF is extracted from the program source code. This function is then applied to find actors using a hill climbing search algorithm. The actors communicate via asynchronous invocations. We have implemented the AOPR approach and applied it in two case studies. The resulting actors can reside either on a cluster running as separate processes or on a multiprocessor computer running as Java threads. The results of our measurements show that AOPR can be applied to improve the performance of legacy software.

This is ongoing research and we are working to extend AOPR to consider hardware constraints such as the number of computational nodes, the number of processors installed on them and network topology during the partitioning.

References

Agha, G., Thati, P., 2004. An algebraic theory of actors and its application to a simple object-based language from object-orientation to formal methods. *LNCS*, **2635**:26-57. [doi:10.1007/b96089]

Andolfi, F., Aquilani, F., Balsamo, S., Inverardi, P., 2000. Deriving Performance Models of Software Architectures from Message Sequence Charts. Proc. 2nd Int. Workshop on Software and Performance, p.47-57.

Bal, H.E., Kaashoek, M.F., 1993. Object Distribution in ORCA Using Compile-Time and Run-Time Techniques. ACM Conf. on Object-Oriented Programming Systems, Languages and Applications, p.162-177.

Chan, B., Abdelrahman, T.S., 2004. Runtime support for the automatic parallelization of Java programs. *J. Supercomput.*, **28**(1):91-117. [doi:10.1023/B:SUPE.0000014804.20789.21]

Deb, D., Fuad, M., Oudshoom, M.J., 2006. Towards Autonomic Distribution of Existing Object Oriented Programs. IEEE Int. Conf. on Autonomic and Autonomous System, p.17-17. [doi:10.1109/ICAS.2006.61]

Fahringer, T., Jugravu, A., 2002. JavaSymphony: New Directives to Control and Synchronize Locality, Parallelism, and Load Balancing for Cluster and GRID-Computing. Proc. joint ACM-ISCOPE conf. on Java Grande, p.8-17. [doi:10.1145/583810.583812]

Gourhant, Y., Louboutin, S., Cahill, V., Condon, A., Starovic, G., Tangney, B., 1992. Dynamic Clustering in an Object-Oriented Distributed System. Proc. OLDA-II (Objects in Large Distributed Applications), p.17-27.

Grimshaw, A.S., 1993. Easy-to-use object-oriented parallel processing with Mentat. *IEEE Trans. Parall. Distr. Technol.*, **26**:39-51.

Hyunsang, Y., Suhyeon, J., Eunseok, L., 2007. Deriving Queuing Network Model for UML for Software Performance Prediction. 5th Int. IEEE Conf. on Software Engineering Research, Management and Application, p.125-131.

Joao, M., Simon, T., Jense, B.J., Oscar, R., 2007. Designing Tool Support for Translating Use Cases and UML 2.0 Sequence Diagrams into a Colored Petri Net. IEEE Proc. 16th Int. Workshop on Scenarios and State Machines.

Maani, R., Parsa, S., 2007. An Algorithm to Improve Parallelism in Distributes Systems Using Asynchronous Calls. 7th Int. Conf. on Parallel Processing and Applied Math, p.312-317.

Mitchell, B.S., Spiros, M., 1999. Bunch: a Clustering Tool for the Recovery and Maintenance of Software System Structure. Proc. IEEE Int. Conf. on Software Maintenance, p.50-59.

Mitchell, S., 2002. A Heuristic Search Approach to Solving the Software Clustering Problem. PhD Thesis, Drexel University, Philadelphia, USA.

Parsa, S., Bushehrian, O., 2005. The design and implementation of a tool for automatic software modularization. *J. Supercomput.*, **32**(1):71-94. [doi:10.1007/s11227-005-0159-5]

Philippsen, M., Zenger, M., 1997. JavaParty transparent remote objects in Java. *Concurr.: Pract. & Exper.*, **9**(11): 1225-1242. [doi:10.1002/(SICI)1096-9128(199711)9:11<1225::AID-CPE332>3.0.CO;2-F]

Robert, G., Hassan, G., 2007. Analyzing Behavior of Concurrent Software Designs for Embedded Systems. IEEE Proc. 10th Int. Symp. on Object and Component-Oriented Real-Time Distributed Computing.

Schoeberl, M., 2006. A Time Predictable Java Processor. Proc. Conf. on Design, Automation and Test in Europe, p.800-805.

Sobral, J.L., Proenca, A.J., 1999. Dynamic Grain-Size Adaptation on Object-Oriented Parallel Programming the SCOOP Approach. Proc. 13th IEEE Int. Symp. on Parallel Processing, p.728-732.

Tahvildari, L., Kontoglannis, K., Mylopoulos, J., 2003. Quality-driven software re-engineering. *J. Syst. Software*, **66**(3):225-239. [doi:10.1016/S0164-1212(02)00082-1]

Zima, H., Chapman, B., 1991. Supercompilers for Parallel and Vector Computers (1st Ed.). Addison Wesley, MA, USA.