Journal of Academic and Applied Studies
Vol. 2(4) April 2012, pp. 22-33
Available online @ www.academians.org
ISSN1925-931X

Automatic Conversion of Remote Invocations in Optimization of Distributed Codes

Alireza Khalilipour, Moharram Challenger

Department of Computer Science, Mahshahr Branch, Islamic Azad University, Mahshahr, Iran

Department of Computer Science, Shabestar Branch, Islamic Azad University, Shabestar, Iran

Abstract

In this paper, an approach is proposed for detecting remote invocations within components of an object oriented distributed system and converting these synchronous invocations to asynchronous ones automatically, without programmers' interference, using CORBA middleware. As a result of this conversion, after invocation of a service in a remote object, it continues execution until it needs the return result of the invocation. In other words, as an advantage, the objects in this system can work asynchronously and collaborate with each other. Another advantage of the proposed approach is that with manipulating the source code of components in required points, it provides a mechanism for the caller object to get the return result of a remote invocation asynchronously from a suitable place without losing entire system's time. This approach can be used in the distributed code generation and optimization tools. This is because these tools also require, in the generation level, an insertion or conversion of communication mechanism in distributed codes.

Keywords: Automatic Conversion, Asynchronous Remote Invocations, Code Optimization

I. Introduction

Nowadays, from the software engineering point of view, central applications are going to be replaced with distributed ones. On the other hand, big organizations and companies have found out that their organization's IT foundation is in fact a distributed computing system. Thus, by using a distributed system, the valuable information resources will become integrated and be used more efficiently.

In these systems, distributed application's components calling each other's services meet systems needs. In other words, one of the important parameters for improving the performance of distributed systems is improving its invocations. Sometimes due to ignorance of developers or misuse of invocations, they make the communication of components synchronously. In this way, a component is blocked with calling a service until the return result is received, even if it does not need the result immediately.

As we can see, to increase the performance, there is a need for another type of communicating mechanism. One possible solution for this problem is replacing asynchronous invocations instead of synchronous ones. However, this solution needs the programmers to be educated with details of such complex issues. Moreover, utilization of this solution is not always possible, since it is not available in all of the tools.

Journal of Academic and Applied Studies
Vol. 2(4) April 2012, pp. 22-33
Available online @ www.academians.org
ISSN1925-931X

One practical solution is that the remote invocations are determined and converted to asynchronous ones automatically. The approach which is offered in this paper works in the similar way and proposes an asynchronous communication mechanism. Moreover, a possible application of the new approach is in tools and environments which generate distributed codes for sequential ones (Mohammad and Micheal, 2001; Galen and Michael, 1999).

This paper is organized as follows: In section 2, the related work is discussed. Pros and cons of asynchronous invocation and the main idea of the paper are elaborated in section 3. Next, in section 4, proposed architecture is presented. Finally, the evaluation of the new approach and conclusion are stated in section 5 and 6 respectively.

II. Related Work

There is an accepted principle that developing a distributed program is always more difficult than developing a sequential one which does the same thing (Djilali, 2003). Thus, the process of automatic conversion of a sequential program to a program which is run-able on a distributed computing environment is ever an open problem.
For automatic conversion of a sequential code to a distributed one, the program is partitioned into several clusters of classes (Zima, 1990) based on communication model of the classes. This is fulfilled using some available tools such as (Galen and Michael L, 1999). With partitioning a sequential program to sets of separated classes, some of the invocations will be converted to intra cluster invocations and some others will be unchanged, see Figure 1.

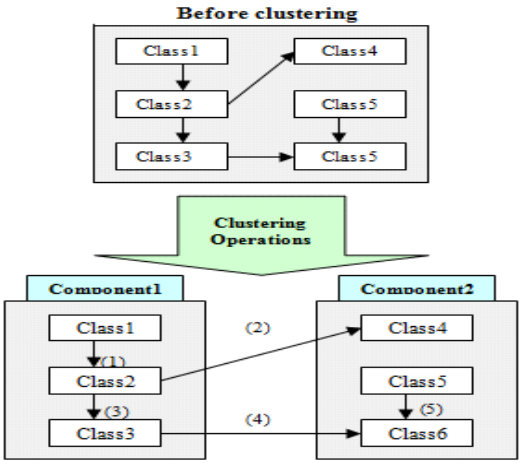

Figure 1: The program before and after clustering

As it is depicted in Figure 1, invocations 1, 3, and 5 remained inter cluster invocation after clustering (decomposition), while, invocations 2 and 4 are converted to intra cluster invocations. Clearly, gained clusters at this level are not capable of being executed over a network and should

be prepared for running on a distributed system by defining a proper communication mechanism and converting intra cluster invocations to the remote invocations.

There are several studies in this regard. One of the simplest solutions is RPC model. This model which is used in old distributed systems such as DCE (Tanenbaum and Steen, 2002) is also a base approach for method invocation from remote objects in some of the object oriented middle wares such as CORBA (OMG, 2012) and DCOM (Platt, 2003).

Globe is another middle ware which uses this approach as its communication mechanism for distributed objects and this is the only way of its communication(Steen et al., 2000). The problem with base RPC is that during method invocation, waiting for the result is inevitable even if the method does not need the result. It is clear that the performance will be the least when called method has no result or caller method does not require the results immediately.

To improve the performance of distributed programs, OneWay communication mechanism has been proposed (Tanenbaum and Steen, 2002). In this mechanism, with accomplishment of a remote call, there is no return result.

JavaSymphony is also a middleware for providing distributed programs in Java (Lorch and Kafura, 2002) which offers this mechanism to the developers. Due to specific application of this mechanism, it is not always possible to use it as communication mechanism for converting sequential code to distributed one (Djilali, 2003). Nevertheless, this mechanism has high performance for the invocations which has no return result.

One of the solutions for improving the performance of distributed programs in most of the cases is using asynchronous invocations. In this kind of communication mechanism, an invocator object continues its execution after invocation until it needs the result. There are many studies and solutions in this regard. For instance, CORBA has introduced a type of invocation called "deferred synchronous request" (Tanenbaum and Steen, 2002) that is special for asynchronous invocations and is supported in late versions of CORBA. In this type of invocation, control mechanism of return result is in lower layer's hand, and that is in fact core ORB of CORBA. Therefore, requester or client has no role in this complex process and it only receives the result from ORB with a suitable call when it needs the return result. CORBA offers another opportunity for developers to make distributed programs asynchronous is utilizing messaging service. In this service, there are two types of communication mechanism: CallBack and Polling (Tanenbaum and Steen, 2002). This capability is also available in .Net (Richter, 2002).

These mechanisms let developer use asynchronous invocations, but they have been designed for situations in which service provider or user objects are running temporarily and are not available for ever. In other words, using these techniques in permanent objects will have overload.

The Microsoft used to apply only standard RPC as communication mechanism of its distributed systems. However, it provided asynchronous invocation for developers in Windows 2000 along with latest version of DCOM (Chappel, 2002; Platt , 2003).

JavaSymphony also provides an asynchronous invocation mechanism (Symphoney, 2012). The caller object, after an asynchronous remote call, receives a handle object to pursue the return result and continue execution. When the caller needs the return result of invocation, it can access the result using the methods which are provided by the handler. If the expected result is available, it will be used; otherwise, the caller will be blocked with the handle until the remote invocation is accomplished and the result is returned.

Journal of Academic and Applied Studies
Vol. 2(4) April 2012, pp. 22-33
Available online @ www.academians.org
ISSN1925-931X

Since using these approaches would not be logical as they needs programmer's knowledge and also has other problems such as low performance, applying them in tools of converting non-distributed code to distributed code or tools of distributed code optimization.

III. Advantages of Asynchronous Invocations

Standard RPC has the most performance when it does a remote procedure call and immediately needs the return result of the invocation. Otherwise, the caller process is blocked and this leads to performance drop. Also, One-way invocations are used only when the remote procedure call or method invocation of a server object has no return result.
There is still another situation in which a service receiver invokes a method of remote object and the method has return result but service receiver does not need the result at the time of invocation and will need it later. This situation is illustrated in figure 2.

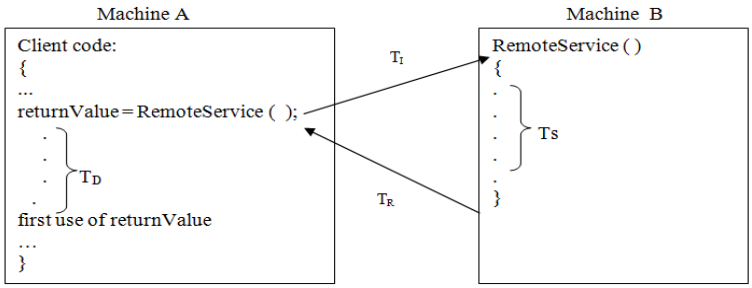

Figure 2: The interval of method call and using return result

In Figure 2, a variable is considered for each of the steps. These variables are:
TS: the time for execution of the called service
TI: the estimated time for remote service call
TR: the return time of call which is almost equal to TI
TD: the time interval between points of the calls and the first point in which the result is used

If this invocation is done in synchronous way, the client and the server are not concurrent. Then, the total time from invocation to first use of the return result is:

$$T_{final-synchronous} = T_I + T_S + T_R + T_D \quad (1)$$

$$T_I \approx T_R \Rightarrow T_{final-synchronous} = T_S + T_D + 2T_I \quad (2)$$

This time will be gained, in case the standard RPC mechanism is used for remote object's communication. The above mentioned time is not optimal and efficient, because in most of the cases, for example in the aforementioned code, it is possible to make client and server concurrent and hence omit *TS* or *TD*.
If there is a mechanism with which the client continues execution immediately after invocation, we can obtain the optimal time. In this case, *TS* and *TD* can be run simultaneously and waiting

Journal of Academic and Applied Studies
Vol. 2(4) April 2012, pp. 22-33
Available online @ www.academians.org
ISSN1925-931X

time will be minimum (even it can be zero when *TD*>*TS*). This mechanism is called Asynchronous with which the total invocation time is:

$$T_{final-asynchronous} = Max(T_S + 2T_I, T_D) \quad (3)$$

Thus, the execution time of above mentioned code, using asynchronous invocations, is either TD or *TS* + 2*TI*.
Comparing the two T_{final} times shows that $T_{final\text{-}Asynchron} < T_{final\text{-}synchron}$ which leads to a performance enhancement in the distributed systems.

A. Main Idea: Separation of Invocation Mechanism from Result Receive

The first step in the separation of service providers and receivers is that a mechanism should be offered to make their processes' exchange the information using an interface, instead of having a permanent linkage. Since the only way for communication of objects in a distributed system is invocation of each other's services, the proposed approach offers an interface with which a caller object can deliver its request to the server object. This mechanism, connecting through an interface, is illustrated in Figure 3.

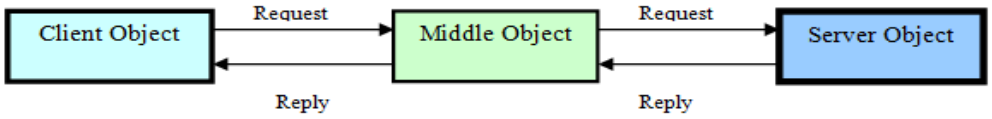

Figure 3. Separation of objects in the client and the server

It is clear that using this mechanism coherency between the client and the server objects will be eliminated. Regarding to the interface, it should be sensitive and provoker; thus, the event service of CORBA is used.

B. Interface Object: CORBA Event Channel

According to the new approach, the client delivers its request, which is service invocation from a server, in the form of an event to the interface. The interface, receiving an event, is triggered and triggers the related server, in the other side, to run the required service. Using this approach, the trigger of the server is done via an interface in the similar way. In other words, in the server side, the interface after receiving an event informs the server object. The event, according to the protocol between the client and the server, includes some information such as parameters required for invocation of the service. For this regard, in the new approach, CORBA channels are used for interface.
CORBA is an architecture for implementing object oriented distributed systems (Wang and Qian, 2005). In this architecture, the objects are communicating with a core which is interface between local and remote objects. CORBA provides some services for distributed system developers with which they can implement and improve the performance of the system. One of these services is event service (Henning and Vinoski, 2003). The importance of this service is

Journal of Academic and Applied Studies
Vol. 2(4) April 2012, pp. 22-33
Available online @ www.academians.org
ISSN1925-931X

that the producer side of the event can continue its normal execution after sending the event to the channel. In fact, the producer does an invocation through event channel, but does not wait for the result. This means that in a system based on CORBA event service, producer and consumer processes are entirely separated and have no direct effect on each others' execution.
The new approach uses this attribute for its asynchronous communications and fulfills the remote invocations between objects in a distributed system using event channels. Using this technique, the client and the server objects, in an object oriented distributed system, is separated from each other and the coherency, which is available in the other approaches during service invocation, is removed. In the result, using the proposed approach, the objects can collaborate asynchronously when it is required.

IV. The Architecture for Asynchronous Invocation

Objects in a distributed system need to call methods of each other to reach the system's goal. The mechanism of asynchronous invocations is introduced in this chapter. To do this, the architecture of the approach is offered in this part. The architecture uses two classes (List 1.a and 1.b), as follows, which will be converted to the server and the client objects in run time.

```
class client {
      public  clientMethod( ) {
            int i,j,k;
            ServerClass Obj;
            k=Obj.remoteMethod(i,j); // calling remote
method
            …
            k++; // first use of   k
            …
       }//end of clientMethod
}//end of class client
```

(a)

```
class serverClass{
      public int remoteMethod(int i, int j)
         {
              return  i*j;
         }// end of method m()
    }//end of serverClass
```

(b)

List 1. (a) Client code (b) server code

In the client class, an instance of the server class is created and then with calling one of its methods, the returned result is used. With the assumption of the ability of conversion, the conversion is fulfilled with the proposed architecture. In this approach, as it can be seen in the architecture illustrated in Figure 4, several components, as auxiliary components, are added to the base classes. Using these auxiliary classes and CORBA event channels, parameters are transferred to the remote objects and as a result remote asynchronous invocations can be done. In other words, these components have the responsibility of managing invocations from the beginning to the end.

Journal of Academic and Applied Studies
Vol. 2(4) April 2012, pp. 22-33
Available online @ www.academians.org
ISSN1925-931X

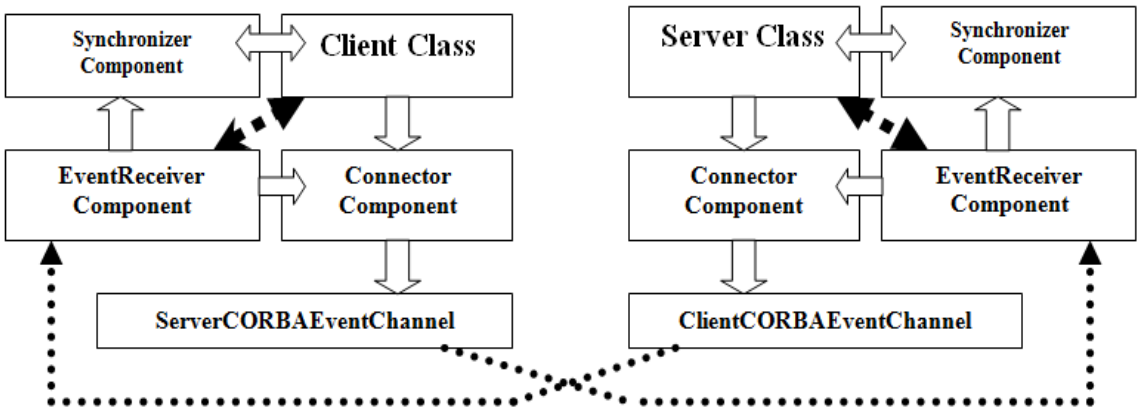

Figure 4. The proposed architecture to make invocations asynchronous

As it can be seen from the architecture, auxiliary components are added to both sides, the client and the server. These components are:

- *ConnectorComponent:* Packing and sending the events to the target channels
- *EventReceiverComponent:* Receiving, Unmarshaling, and detecting packets or events
- *SynchronizerComponent:* Managing returned results from remote objects

A. The Mechanism: Overall

The client asks from *ConnectorComponent* for invocation of a service. Hereby, it delivers the required parameters for invocation to *ConnectorComponent* which sends it to the event channel of server after marshalling (packing). Thus, the first change or conversion in the initial code of the client is converting the invocation to calling a service in *ConnectorComponent*. The client, after invocation of *ConnectorComponent*, continues execution. To do this, ConnectorComponent provides SendEvent interface for the client.

With calling the SendEvent service from *ConnectorComponent*, the component marshals the received parameters as a CORBA compatible event and sends it to the server's event channel. In the other side, *EventReceiverComponent* receives a packet or an event and extracts required parameters for service invocation after decomposing. Required parameters for invocation include a reference of the object, method name, and the arguments for that method. In this way, *EventReceiverComponent* can fulfill the invocation.

Doing the invocation, if the service has a result, it should be returned to the client object. At this point, *EventReceiverComponent* requests from *ConnectorComponent* to marshal the result and insert it into the event channel of the source object (client). For this purpose, *ConnectorComponent* has ReplyEvent service. At the time of returning the result to the client's event channel, *EventReceiveComponent*, in client side, is triggered and receives the return result. This result will be used in a line of client object which *EventReceiverComponent* is not aware of its exact line number. For this reason, it should save it in a place where the client can take and use it. This duty, saving the return result, is for *SynchronizerComponent* which has a table for return results. This component saves the result in a row of the table which was created before the invocation. In this table, each invocation has a unique ID with which it prevents the conflict.

Journal of Academic and Applied Studies
Vol. 2(4) April 2012, pp. 22-33
Available online @ www.academians.org
ISSN1925-931X

Finally, when the client object needs the result, asks it from *SynchronizerComponent*. Clearly, the related code of the client object should be changed which is discussed in next section.

B. Steps of Asynchronous Remote Invocations in the New Approach

Overall, the steps of an asynchronous invocation in the proposed architecture include (based on the client and server class introduced in previous section):

- The client creates a unique ID for invocation and a new row in the result table via *SynchronizerComponent*. Then, it delivers the invocation request, including required parameters, to *ConnectorComponent*. Thus, the initial code of the client will be changed as follows(List 2.a and List 2.b):

```
class client {
    public  clientMethod() {
       int I,j,k;
       ServerClass O;
       i=O.m(k,j); //calling remote method
        …
     }
  }
```

(a)

```
class client {
    public  clientMethod() {
       int I,j,k;
       ServerClass O;
       Callid=CreateNewID();
       SynchronizerComponent.Insert(Callid);
       ConnectorComponent.SendEvent("serverClass",
            ObjectRef, "methodname", parameters)
       // i=O.m(k,j); // deleted by translator engine
        …
     }
}
```

(b)

List 2. (a) client class before conversin (b) after conversion

- *ConnectorComponent*, receiving the parameters, marshals them in a CORBA event and inserts it in the server's event channel. This operation is done via SendEvent service of *ConnectorComponent*. Thus, part of this component will be produced as follows (List 3):

```
class ConnectorComponent{
      Public SendEvent( classname, objReference, methodname, parameters){
            CORBA::Any  Event;
            Event= Marshall(Parameters); // with specific algorithm
            ServerEventChannel.Push(Event);
      } //end of SendEvent
}
```

List 3. Generating ConnectorComponent

- In the other side, *EventReceiverComponent* receives the packet and after decomposing, extracts required parameters. Next, for invocation on the server object, it requests the call from its *ConnectorComponent* to return its result to the client object's event channel and to this end, ReplyEvent of *ConnectorComponent* is used. Therefore, part of the code for this component is as follows (List 4):

```
Class EventReceiverComponent{
```

```
        Public EventReceiverComponent() {
              CORBA::Any Event, Result;
              Event=ServerEventChannell.Pull();
              // UnMarshall the packet, with the specific algorithm
              Result= ObjRef->Methodname(Parameters);
              ConnectorComponent.ReplyEvent(Result);
         }// end of EventReceiverComponent
}
```

List 4. Generating EventReceiverComponent

- *ConnectorComponent*, in the server side, marshals the return result and then sends it to client object's event channel. So, ReplyEvent service is produced as follows (List 5):

```
class ConnectorComponent{
      public    ReplyEvent(CORBA::Any  Result){
                  CORBA::Any Event;
                  //Marshalling Return value and CallId to a packet such as EVENT
                  ClientEventChannel.Push(Event);
       }// end of ReplyEvent
}
…..
```

List 5. Generating ConnectorComponent

- At this step, the returned result has been entered into the client's event channel. Thus, *EventReceiverComponent* receives it and after decomposition and detection of its type, saves it in the result table. This is done via *SynchronizerComponent* (List 6).

- *SynchronizerComponent* saves the returned result in the row of the table which is created special for it.

```
class EventReceiverComponent{
         Public EventReceiverComponent ( ){
               CORBA::Any Event;
               Event = ClientEventChannel.Pull();
               //UnMarshall the event to getting CallId and return value
               SynchronizerComponent.Update(ReturnValue, CallId);
          }
          ….
}
```

List 6. Generating EventReceiverComponent

- There should be some changes in the line of client's initial code in which the result is used. That is because the result is in the result table and its management is *SynchronizerComonent*'s responsibility. These changes, according to the example of previous section, is as follows (List 7.a and 7.b):

Journal of Academic and Applied Studies
Vol. 2(4) April 2012, pp. 22-33
Available online @ www.academians.org
ISSN1925-931X

```
class client {
      public  clientMethod( ) {
           int I,j,k;
           ServerClass O;
           i=O.m(k,j); // calling remote method
           …
           …
           i++; // first use of   i
           …
           }//end of clientMethod
}//end of class client
```

(a)

```
class client{
    …..
    …..


    CORBA::Any RV;
    RV= SynchronizerComponent.update(Callid);
    RV>> i;
    i++;
    …
  }//end of clientMethod
}//end of class client
```

(b)

List 7. (a) before conversion (b) after conversion

To implement the code conversion engine according to the mentioned algorithm, it is assumed that initial classes' descriptions are based on architecture description languages(Magee et al., 1995; Jonkers, 2001; Aldrich et al., 2002; Luckham et al., 1995) for doing the changes on initial codes. Some of the required specifications are:

- Synchronization points (the line number of the code in which invocation is done)
- The properties of remote service (including name, name of the owner class, and names and types of the parameters)
- The line number in which the result is used for the first time

V. Performance Evaluation and Speed of Distributed Code

To evaluate the performance and speed of converted code and also to compare it with the initial code, two services which have relatively high computations are used. These services are examined based on their execution time for both initial code and converted code. The services calculate the maximum saddle and invert saddle points of a matrix. The methods for finding these points are considered in separate classes. The "finding_Saddle_points" class is used to find the maximum saddle points and the "finding_invert_Saddle_points" class is used to find invert saddle points. In the distributed version, which is result of the conversions, the processes of calculating maximum saddle points and invert saddle points are done simultanously and the point of returning the result is as synchronizing point of distributed code. To compare the performance of initial and improved codes, the programs have been executed separately and their running times are compared. As it is expected, the result shows significant improvement in the speed of converted code versus initial code, see Table I.

TABLE I
THE RUN TIME RESULTS OF EXECUTING INITIAL AND CONVERTED CODES

Matrix Dimensions	10*10	25*25	50*50	70*70	90*90	100*100	150*150	181*181

Runtime for initial code (second)	0.2	1.12	5.2	9.1	15.6	25.8	39.4	60.7
Run time for converted code (second)	0.33	1.17	2.6	3.6	5.3	10.9	17.5	26.8

It worth noticing that Mico implementation version (Puder and Romer,1998; Mico, 2012; MicoCCM, 2012) is used as middle ware. The results of Table 1 is manipulated as the diagram of Figure 5 which shows that with increasing the dimensions of the matrixes, representing the problem size, the performance of distributed code increases as well.

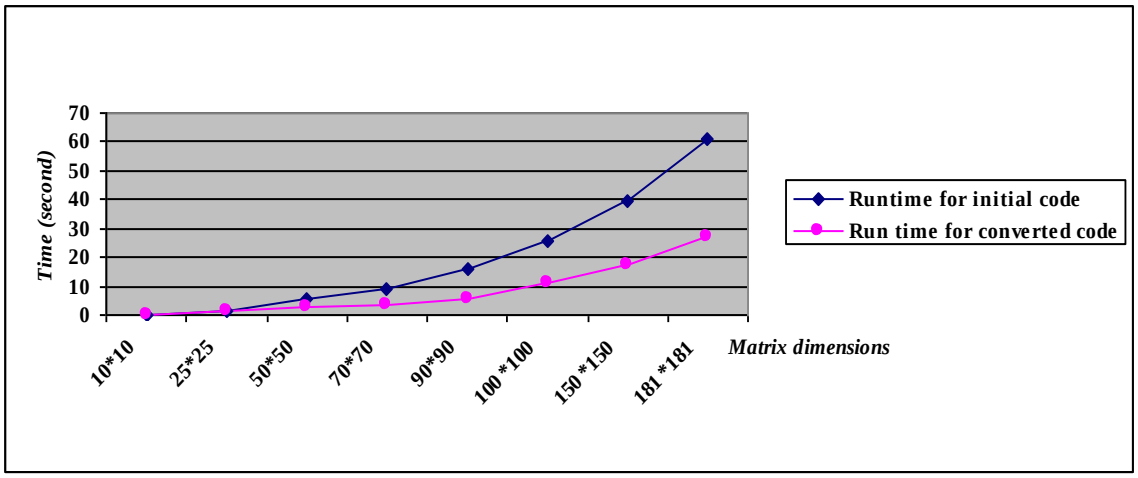

Figure 5. The diagram of comparing the run times for execution of initial and converted codes

VI. Conclusion

In this paper, a new approach is proposed for automatic conversion of synchronous invocations to asynchronous calls. The main goal of this paper is increasing performance of the distributed codes with improving invocations type. To this end, in the new approach, having server and client's initial codes along with synchronization points in client code and using CORBA event service, the code conversion and generation are fulfilled. The code generation is constituted of producing some components with which invocation and return are realized. In fact, the code generation is applied for the points of the client's code where the code analyzer detects them as appropriate for conversion. To separate the client and the server and to make them parallel, an event channel is used from which the invocation's required parameters and the return results are exchanged. However, the limitation for this approach is the need for synchronization points to handle the initial code.

As a future work for this study, it possible to add another component and use multi-threading technique to have parallelism in client side. In this way the server will be exempt from dealing with event channel. Thus, the server will have no change.

References

Aldrich J, Chambers C, and Notkin D (2002). ArchJava: Connecting Software Architecture to Implementation. *Proceedings ICSE. IEEE Computer Society, Los Alamitos, CA.*

Chappel D (2002). Understanding .NET, Addison-Wesley.

Djilali S (2003). P2P-RPC:Programming Scientific Applications on Peer-to-Peer Systems with Remote Procedure Call. *Proceedings of the 3th IEEE International Symposium on Cluster Computing and the Grid.*

Galen C and Michael L (1999). The Coign automatic Distributed PartitioningSystem. *Proceedings of the 3th Symposium on Operating Systems Design and Implementation.*

Henning M, Vinoski S (2003). Advanced CORBA Programming with C++. Addison-Wesley.

Jonkers H (2001). Interface-centric Architecture Descriptions. Proceedings. *The Working IEEE/IFIP Conference on Software Architecture (WICSA), IEEE Computer Society, Los Alamitos, CA.*

Lorch M and Kafura D (2002). Symphony – A Java-based Composition and Manipulation Framework for Computational Grids. *Proceedings of Grid computing.*

Luckham D, Kenney J, Augustin L, Vera J, Bryan D, and Mann W (1995). Specification and Analysis of System Architecture Using Rapide. *IEEE Transactions on Software Engineering, Special Issue on Software Architecture,* 21(4), pp 336-355.

Magee J, Dulay N, Eisenbach S and Kramer J (1995). Specifying Distributed Software Architectures. *Proceedings of 5th European Software Engineering Conference (ESEC '95),* LNCS 989, pp 137–153.

Mico. <http://www.mico.org/Mico> (Last access: May 2012).

MicoCCM. <http://www.fpx.de/MicoCCM> (Last access: May 2012).

Mohammad M and Micheal J (2001). Adjava-automatic distribution of Javaapplications. *Proceedings of the 25th Australasian Computer Science Conference in research and Practice,* Vol. 4.

Morgan, C. (1994) Programming from Specifications. Prentice Hall, Englewood Cliffs, NJ.

OMG. <http://www.omg.org/corba> (Last access: May 2012).

Platt D (2003). Introducing Microsoft .NET. Microsoft Press.

Puder A, Romer K (1998). Mico is CORBA.dpunkt verlag.

Richter J (2002). Applied Microsoft .NET Framework Programming. Microsoft Press, Redmond, WA.

Steen M, Homburg P and Tanenbaum A (2000). Globe: A Wide-Area Distributed System. *IEEE Concurrency, Vol 7*

Symphoney. <http://www.symphony.cs.vt.edu> (Last access: May 2012).

Tanenbaum A and Steen M (2002). Distributed Systems: Principles and Paradigms. Prentice Hall, Upper Saddle River, NJ.

Wang A, Qian K (2005) Component-oriented programming. Southern Polytechnic State University Marietta, Georgia, A John Wiley & Sons, INC., Publication, Hoboken, New Jersey.

Zima H (1990). Super-compilers for Parallel and Vector Computers. ACM Press.