Paper presented in 8th International Conference on Information Technology, Bhubaneswar, India Dec 20-23 2005 Archived in Dspace@nitr, http://dspace.nitrkl.ac.in/dspace

A Parallel Algorithm for Dynamic Slicing of Distributed Java Programs in non-DSM Systems

Durga Prasad Mohapatra
Department of CSE
National Institute of Technology
Rourkela, Orissa, 769008
durga @nitrkl.ac.in

Rajib Mall
Department of CSE
Indian Institute of Technology
Kharagpur, WB, 721302
rajib@cse.iitkgp.ernet.in

Rajeev Kumar
Department of CSE
Indian Institute of Technology
Kharagpur, WB, 721302
rkumar@cse.iitkgp.ernet.in

ABSTRACT
We propose a parallel algorithm for dynamic slicing of distributed Java programs. Given a distributed Java program, we first construct an intermediate representation in the form of a Distributed Program Dependence Graph (DPDG). We mark and unmark the edges of the DPDG appropriately as and when dependencies arise and cease during run-time. Our algorithm can run parallely on a network of computers, so that each node in the network contributes to the dynamic slice by computing its local portion of the global slice in a fully distributed fashion.

Keywords
program slicing, program dependence graph, distributed programming.

1. INTRODUCTION
As software applications grow larger and become more complex, program maintenance activities such as adding new functionalities, porting to new platforms, and correcting the reported bugs require enormous effort in software development. This is especially true for distributed programs. In order to cope with this scenario, programmers need effective computer supported methods for decomposition and dependence analysis of programs. Program slicing is one method for such decomposition and dependence analysis. A program slice with respect to a specified variable at some program point consists of those parts of the program which potentially affect the value of that variable at the particular program point [1]. A static slice is valid for all possible executions of a program while a dynamic slice considers only a particular execution of a program [2]. Program slicing has been found to be useful in a variety of applications such as debugging, program understanding, testing and maintenance, etc. [3-6].

It is usually accepted that understanding and debugging of distributed object-oriented programs are much harder compared to those of sequential programs. The non-deterministic nature of distributed programs, the lack of global states, unsynchronized interactions among threads , multiple threads of control and a dynamically varying number of processes are some reasons for this difficulty. An increasing number of resources are being spent in debugging, testing and maintaining these products. Slicing techniques promise to come in handy at this point. Through the computation of a slice for a message passing program, one can significantly reduce the amount of code that a maintenance or development engineer has to comprehend or analyze to achieve some maintenance tasks.

In this paper, we propose a parallel algorithm for computing dynamic slices of distributed Java programs. We have named our proposed algorithm parallel dynamic slicing (PDS) algorithm for distributed Java programs in. To achieve faster response time, our algorithms can parallely run on several machines connected through a network, rather than running it on a centralized machine. In order to run the algorithm parallely, we use local slicers at each machine.

2. BASIC CONCEPTS
A distributed object-oriented program $P = (P_1, P_2,\ldots, P_n)$ is a collection of concurrent individual programs Pi such that each of the Pi's may communicate with other programs through the reception and transmission of messages.

A distributed control flow graph (DCFG) G of a component program Pi of a distributed program $P = (P_1, P_2,\ldots, P_n)$ is a flow graph (N, E, Start, Stop), where each node $n \in N$ represents a statement of Pi, while each edge $e \in E$ represents potential control transfer among the nodes. Nodes *Start* and *Stop* are unique nodes representing entry and exit of the component program P_i respectively. There is a directed edge representing a control flow from node a to node b if control may flow from node a to node b.

The intermediate representation for a concurrent object-oriented program on a single machine is constructed statically as in [7]. But, for distributed object-oriented programs,
we can have communication dependency between threads running on different machines. A getInputStream() call executed on one machine, might have a pairing getOutputStream() on some other remote machine. To incorporate this aspect, we introduce a logical(dummy) node in the DPDG. We call this logical node as a C-node. In the following, we define the C-node and the intermediate repre-

sentation for distributed Java programs used by our dynamic slicing algorithm. We now describe the role of a C-node.

```
1. class clthd extends Thread {
2.   BufferedReader rcvmsg;
3.   PrintWriter sendmsg;
4.   BufferedReader in=new BufferedReader(new InputStreamReader(system.in));
5.   Socket socket;
6.   public void run() {
7.      socket=new Socket("10.5.18.49",1500);  // connecting to server at given ip and port no.s
8.      sendmsg=new PrintWriter(socket.getOutputStream());  // declaration of sendmsg
9.      rcvmsg=new BufferedReader(new InputStreamReader (socket.getInputStream()));  // declaration
10.     String s=in.readLine();
11.     int x=Integer.parseInt(s);
12.     int z,q,y=15;
13.     if(x>y)
14.        z=x-y;
        else
15.        z=x+y;
16.     sendmsg.println(z);       // sending value of z to server
17.     System.out.println("value of z is: "+z);
18.     String msgfrom_server=rcvmsg.readLine();       // receiving value from server
19.     int p=Integer.parseInt(msgfrom_server);
20.     if(p>x)
21.        q=p-x;
        else
22.        q=p+x;
23.     System.out.println("total is: "+q);
      }
24.   public class client {
25.     public static void main(String args[]) {
26.        clthd t=new clthd();
27.        t.start();
       }
     }
   }
```

Figure 1. An Example Client Program

Let G_{D1} be the DPDG of the component program P_1 and G_{D2} be the DPDG of the component program P_2. Let x be a node representing the statement which invokes a getOutputStream() method, in G_{D1}. Let y be the node representing the statement which invokes the corresponding getInputStream() method, in G_{D2}. A C-Node represents a logical connection of the node y of DPDG G_{D1} with the node x of the remote DPDG G_{D2}. A C-node does not represent any specific statement in the source code of a component program. Rather, it is used to encapsulate the information of the triplet <send_TID, send_node_number, dynamic_slice_at_ send_ node > representing the pairing of the components in a distributed program. Here, send_TID represents the id of the thread sending the message, send_ node_ number represents the particular label number of the statement sending the message and dynamic_slice_at_send_ node represents the dynamic slice at the sending node. Communication dependencies among threads of distinct component programs are captured using the C-nodes. The sending thread passes the message contents to the slicer. The slicer piggybacks this triplet with the actual message. Whenever any thread executes a getInputStream() call, the slicer extracts the triplet from the message in the message queue and passes the actual message to the receiving thread. Thus the slicer updates the information on C-nodes and establishes the communication dependency.

Now, we define a *Distributed Program Dependence Graph* (DPDG). The distributed program dependence graph (DPDG) G_D of the component-program P_i is a directed graph (N_D, E_D) where each node n (excepting the dummy nodes)$\in$ N_D represents a statement in P_i. For x, y $\in$ N_D, (y,x) $\in$ E_D iff any one of the following holds:

```
1. class shar {
2.   int s;
3.   boolean flag=true;
4.   synchronized public void put(int c) {
5.     s=c;
6.     notify();
7.     flag=false;
     }
8.   synchronized public int get() {
9.     if(flag==true)
10.       wait();
11.    return s;
     }
   }
12. class thread1 extends Thread {
13.   BufferedReader rcvmsg;
14.   PrintWriter sendmsg;
15.   Socket serv_socket;
16.   int b,c;
17.   shar obj;          // declaration for shared object
18.   BufferedReader o=new BufferedReader(
        new InputStreamReader(System.in));
19.   public thread1(Socket req,BufferedReader in,
          PrintWriter out,shar ob) {
20.   serv_socket=req;
21.   sendmsg=out;
22.   rcvmsg=in;
23.   obj=ob;
     }
24.   public void run() {
25.     String msg=rcvmsg.readLine();  // receiving message from client prog
26.     int a=Integer.parseInt(msg);
27.     System.out.println("received from client is: "+a);
28.     String mss=o.readLine();
29.     int b=Integer.parseInt(mss);
30.     if(a>b)
31.         c=a-b;
        else
32.         c=a+b;
33.     obj.put(c);   // sending the value of c to thread2
34.     System.out.println("thd1: "+c);
    }       }
35. class thread2 extends Thread  {
36.   BufferedReader rcvmsg;
37.   PrintWriter sendmsg;
38.   Socket serv_socket;
39.   shar obj;   // declaration for shared object
40.   BufferedReader o=new BufferedReader(
        new InputStreamReader(System.in));
41.   public thread2(Socket req,BufferedReader in,
          PrintWriter out,shar ob) {
42.     serv_socket=req;
43.     sendmsg=out;
44.     rcvmsg=in;
45.     obj=ob;      }
46.   public void run() {
47.     int e,g,f=10;
48.     e=obj.get();    // receiving the value from thread1
49.     if(e>f)
50.       g=e-f;
        else
51.       g=e+f;
52.     sendmsg.println(g);    }     }  // sending value of g to client
53. public class syn_server {
54.   public static void main(String args[]) {
55.     ServerSocket serv_socket;
56.     BufferedReader rcvmsg;
57.     PrintWriter sendmsg;
58.     shar obj=new shar();
59.     serv_socket=new ServerSocket(1500);  // creating a new port
60.     Socket socket=serv_socket.accept();   // accepts client
61.     sendmsg=new PrintWriter(socket.
              getOutputStream(),true);  // declaration for sendmsg
62.     rcvmsg=new BufferedReader(new
          InputStreamReader(socket.getInputStream()));  // declaration for rcvmsg
63.     thread1 t1=new thread1(socket,input,output,obj);
64.     thread1 t2=new thread2(socket,input,output,obj);
65.     t1.start();
66.     t2.start();
     } }
```

Figure 2. An Example Server Program

1. y is control dependent on x. Such an edge is called a control dependence edge.
2. y is data dependent on x. Such an edge is called a data dependence edge.
3. y is thread dependent on x. Such an edge is called a thread dependence edge.
4. y is synchronization dependent on x. Such an edge is called a synchronization dependence edge.
5. y is communication dependent on x. Such an edge is called a communication dependence edge.

For all the nodes x, representing getInputStream() calls, in the component program P_i , a dummy node C(x) is created, and a dummy communication edge (x, C(x)) is added.

The DPDGs of the example programs given in Fig. 1 and 2 are shown in Fig. 3 and 4.

3. Parallel Dynamic Slicing (PDS) Algorithm

Before execution of a distributed Java program P, the DCFG of each of the component program P_i is constructed statically. Next, we statically construct the DPDG of each component program P_i by using the DCFG. During execution of a component program P_i ,we mark an edge of the DPDG when its associated dependence exists, and unmark the edge when its associated dependence ceases to exist. Since control dependencies do not change during run-time, we permanently mark the control dependence edges. We consider all the dependence edges excepting control dependence edges for marking and unmarking. In our approach, we allow communication to occur across different machines. So, we perform some additional task to capture this communication. Intermachine communication is captured by adding C-nodes in the DPDG. The addition of C-nodes in the DPDG takes care of any communication dependency that might exist at run-time between communicating threads on different machines.

Now, we define the dynamic slice with respect to the present execution of the statement u, for the variable var, in thread p, as Dynamic_Slice(p, u, var) ={(p, x_1),…, (p, x_k)}∪ Dynamic_Slice(p, x_1, var) ∪…∪ Dynamic_Slice(p, x_k ,var).

Let var_1, var_2, . . . , var_k be all the variables used or defined at statement u in some thread p. Then, we define dynamic slice of the whole statement u as dyn_slice(p, u) = Dynamic_ Slice(p, u, var_1) ∪ Dynamic_Slice(p, u, var_2) ∪…∪ Dynamic_ Slice (p, u, var _k).

Our slicing algorithm operates in three main stages:

1. Constructing the intermediate program representation graph statically
2. Managing the DPDG at run-time
3. Computing the dynamic slice

In the first stage, the DCFG of each component program P_i is constructed from a static analysis of the source code. Also, in this stage using the DCFG the static DPDG is constructed, as the DCFG provides the information regarding the control flow in each of the component program. The stage 2 of the algorithm executes at run-time and is responsible for maintaining the DPDG as the execution proceeds. The maintenance of the DPDG at run-time involves marking and unmarking the different dynamic dependencies as they arise and cease. Stage 3 is responsible for computing the dynamic slices for a given slicing criterion using the DPDG. Once a slicing criterion is specified, our dynamic slicing algorithm computes the dynamic slice with respect to the slicing criterion by looking up the corresponding *Dynamic _Slice* computed during run time.

Working of the PDS Algorithm

Consider the distributed Java program given in Fig. 1 and 2. The threads in the client program and server program are identified by unique thread-ids. Let the thread-id of the *clthd* in Fig. 1 be 1001, the thread-id of *thd1* in Fig. 2 be 2001 and the thread-id of *thd2* in Fig. 2 be 2002. The updated DPDGs are obtained after applying stage 2 of the PDS algorithm and are shown in Fig. 3 and Fig.4. Let us compute the dynamic slice with respect to variable q at statement 23 of the thread *clthd* in the *client program* (Fig. 1). This gives us the slicing criterion < 1001, 23, q >. With input data s = 20 in the client program in Fig.1 and b = 2 in the server program in Fig. 2, we explain how our PDS algorithm computes the dynamic slice.

According to our PDS algorithm, the dynamic slice at statement 23, is given by the expression Dynamic_ Slice(1001, 23, q) ={(1001, 21), (1001, 6)} ∪ dyn_slice(1001, 21) ∪ dyn_slice(1001, 6). Evaluating the expression, we get the final dynamic slice at statement 23 of Fig. 1. The statements included in the dynamic slice are shown as shaded vertices in Fig.3 and 4.

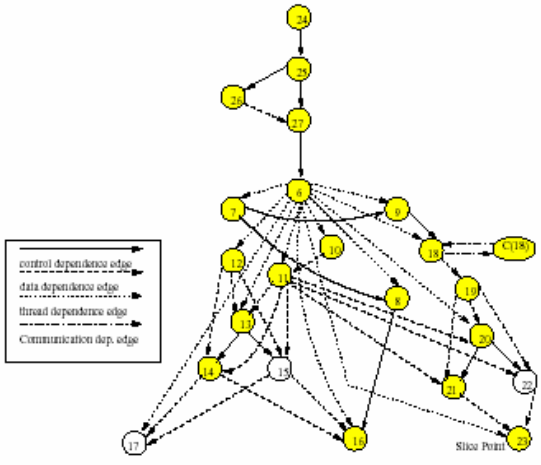

Figure 3: Updated DPDG of Client Program

Modification of the Algorithm for non-DSM Systems:

A distributed system having no support for shared memory reduces to a message passing system and we term them in our study as a non-DSM system. In order to handle non-DSM systems, we have introduced a new type of node, *R-node*, in our intermediate representation DPDG. Because of addition of these logical (dummy) nodes in the DPDG, the above algorithm is updated by adding the functionality for handling of these *R-nodes*. The existence of *R-nodes* in the DPDG depends on how we are maintaining the most recent information on shared variables. We have already discussed extensively how the C-nodes are incorporated in the DPDG. The *R-nodes* are handled in the similar manner for shared variables in non-DSM systems.

The modifications to be done in the above algorithm to incorporate the use of *R-nodes* involve the following steps:

Stage-1: DPDG Construction

For each shared variable var used at u, do
Add a R-node R(u)
Add data dependence edge (u, R(u)) and unmark it.

Stage-2: Managing the DPDG at Run-Time

Update shared data dependencies: For every shared variable *var* used at node (p,u), mark the data dependence edge corresponding to the most recent definition recentDef(p,var) available at the R-node of the variable var.

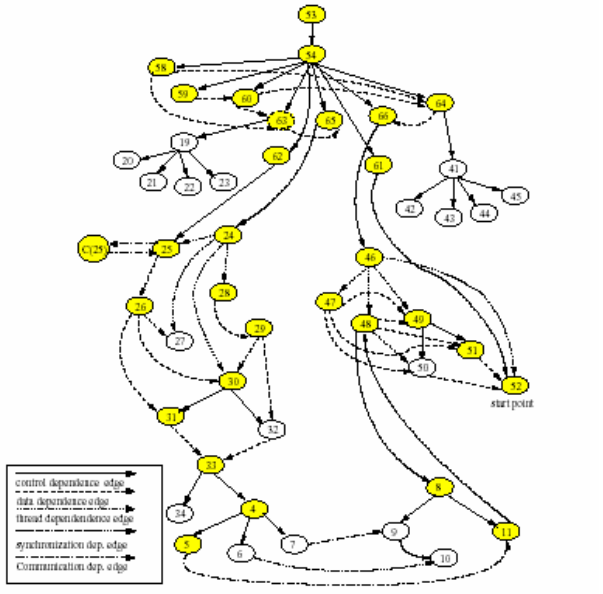

Figure 4: Updated DPDG of Server Program

4. Conclusions

In this paper, we have proposed a novel technique for computing dynamic slices of distributed Java programs in non-DSM systems. We have introduced the notion of *distributed dependence graph* (DPDG). We have named our algo-

rithm *parallel dynamic slicing* (PDS) algorithm. It is based on marking and unmarking the edges of the DPDG as and when the dependencies arise and cease at run-time.

REFERENCES

[1] M. Weiser. Programmers use slices when debugging. *Communications of the ACM*,25(7):446-452, 1982.

[2] B. Korel and J. Laski. Dynamic program slicing. *Information Processing Letters*,29(3):155{163, 1988.

[3] R. Mall. *Fundamentals of Software Engineering*. Prentice Hall, India, 2nd Edition, 2003.

[4] D. Goswami and R. Mall. An efficient method for computing dynamic program slices. *Information Processing Letters*, 81:111-117, 2002.

[5] G. B. Mund, R. Mall, and S. Sarkar. An efficient dynamic program slicing technique. *Information and Software Technology*, 44:123-132, 2002.

[6] X. Zhang, R. Gupta, and Y. Zhang. Efficient forward computation of dynamic slices using reduced ordered binary decision diagrams. In *International Conference on Software Engineering*, 2004.

[7] Durga Prasad Mohapatra, Rajib Mall, and Rajeev Kumar. An efficient technique for dynamic slicing of concurrent Java programs. *In Proceedings of Acian Applied Conference on Computing* (AACC-2004), *Kathmandu, LNCS Springer-Verlag*, volume 3285, Pages 255-262, October 2004.