Preliminary results of a persistent execution engine

Kasper Østerbye

IT University Technical Report Series TR-2006-82

ISSN 1600-6100 January 2006

ISSN 1600-6100

ISBN 87-7949-121-9

Copies may be obtained by contacting:

IT University of Copenhagen
Glentevej 67
DK – 2400 Copenhagen NV
Denmark

Telephone: +45 38 16 88 88
Telefax: +45 38 16 88 99
Web: www.itu.dk

Preliminary results of a persistent execution engine

Kasper Østerbye, kasper@itu.dk
IT University of Copenhagen
January 25, 2006

Abstract: Normally one strives to make a virtual machine as fast as possible. This report documents what is likely to be the slowest virtual machine ever created. The idea behind this VM is to support execution processes rarely do anything at all, but which run for very long real time.

1. Introduction

A workflow process controls how a specific task is carried out in collaboration between a system of computers, and a group of humans. Such a process will most of the time wait for a human agent to finish a subtask, wait for a specific external event, or just wait for a particular duration or until a certain date.

We believe that being able to write such workflows in a programming language, in which one focuses on the description of the individual process, will make it easier to understand and express the processes involved. Our vision is to create a language which does to workflows what Simula did to simulation, providing a language that provides direct support for the material view [Kreutzer, 95]. That is, allow us to focus on the processes and their interconnections, rather than the scheduling of the tasks of the workflow. The individual processes in a simulation spend most of its (simulated) time waiting. As our workflow processes will wait in real time, it is important that the process can be easily persisted when it has to wait.

A very radical (and naïve) approach is to let all state in the virtual machine be persisted at all times. This paper reports on a virtual machine in which each step of the fetch-execute loop of the interpreter is itself a database transaction. This is the most radical approach we can take, which will provide us with a base performance against which we can measure later optimizations.

This research is motivated by the idea of a workflow language. However, workflow languages and their implementation will not be further addressed in this paper.

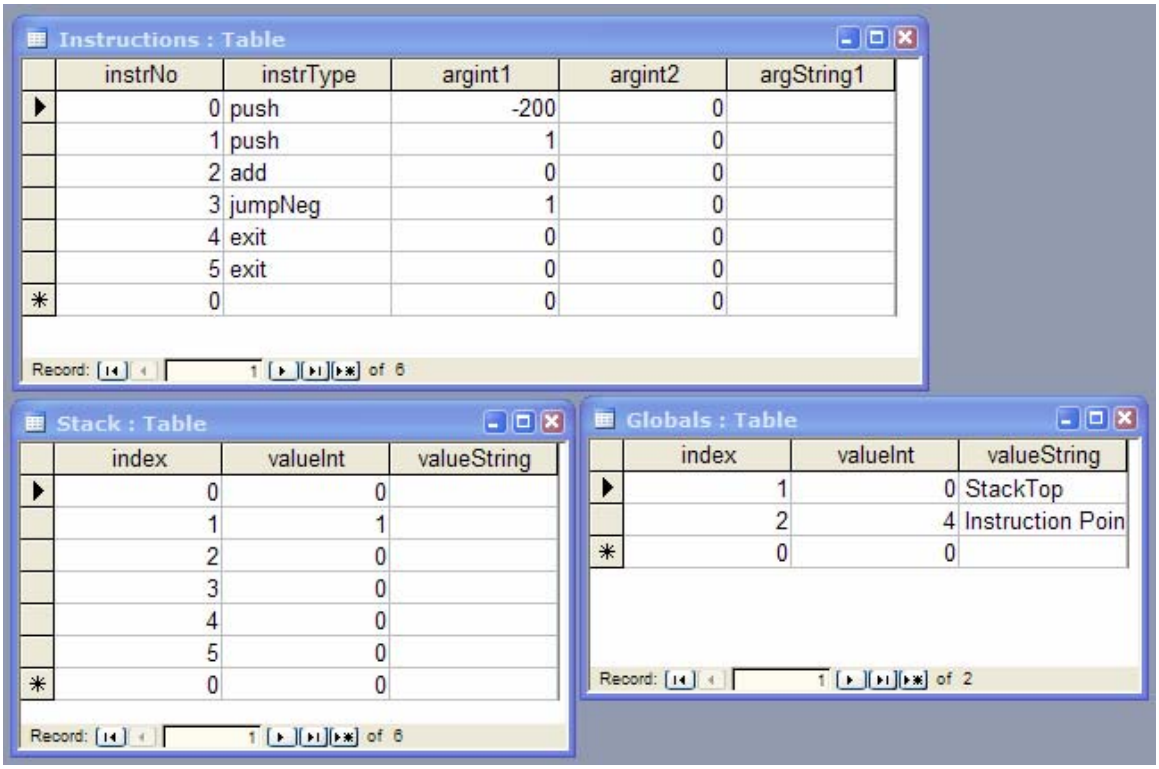
Section 2 will provide a description of the experimental virtual machine. Next, we report on some timing results done using different kinds of queries.

2. Experimental setup

As indicated by the title, we are not anywhere near realizing our vision of a persistent workflow language. Our experiment is a model of a stack machine. It stores the stack, program, and instruction pointer in tables. The only instructions implemented are a push, add, conditional jump, and exit. Nevertheless, those are sufficient to allow us to write a simple loop.

The database we have used is Microsoft Access, primarily because it is easy to set up.

The layout of the used tables can be seen below:



Instructions : Table

instrNo	instrType	argint1	argint2	argString1
0	push	-200	0	
1	push	1	0	
2	add	0	0	
3	jumpNeg	1	0	
4	exit	0	0	
5	exit	0	0	
0		0	0	

Record: 1 of 6

Stack : Table

index	valueInt	valueString
0	0	
1	1	
2	0	
3	0	
4	0	
5	0	
0	0	

Record: 1 of 6

Globals : Table

index	valueInt	valueString
1	0	StackTop
2	4	Instruction Poin
0	0	

Record: 1 of 2

The program used in the tests is encoded in the table Instructions. It mirrors the following simple stack code:

```
start:  push    -200        ; Initial value
loop:   push    1           ; push 1
        add                 ; add one
        jumpneg     loop    ; test stacktop, if negative, jump to loop
        exit                ; end program
```

A.1 Query based results

The first set of interpreters was written in Java using the java.sql classes. Three variations on the interpreter have been tried out.

- *Plain*, where the necessary queries were executed as plain strings sent to the database. The code is shown in appendix A.
- *Prepared*, where the same SQL statements are used, but they are precompiled them as prepared statements. The code is in appendix B.

- *Integrated*, a variation in which we integrated the update of the instruction pointer into the queries themselves. The code is in appendix C.

In all three cases, the instruction fetch execute loop is written as:

```
public int interpret(Statement stmt) throws SQLException{
    int count = 0;
    while(true){
        count++;
        ResultSet instr = stmt.executeQuery(getInstruction);
        instr.next();
        String opcode = instr.getString("instrType");
        if (opcode.equals("push") ){
            stmt.execute(pushIntInstruction);
            stmt.execute(incrIP);
        }else if (opcode.equals("add") ){
            stmt.execute(addInstruction);
            stmt.execute(incrIP);
        }else if (opcode.equals("jumpNeg") ){
            stmt.execute(jumpNegInstruction);
            stmt.execute(incrIP);
        }else if (opcode.equals("exit") ){
            break;
        }else{
            out.println("Unknown opcode: " + opcode);
        }
    }
    return count;
}
```

The result of the interpreter method is the number of instructions executed, which is used for the timing results. The interpreter gets the current instruction, and depending on its type, an instruction is issued and the instruction pointer is executed.

The three instructions are coded as:

```
addInstruction =
        "UPDATE stack AS s1, stack AS s2, Globals AS sp " +
            "SET s2.valueInt = s1.valueInt+s2.valueInt "+
            ",    sp.valueInt = sp.valueInt-1 " +
            "WHERE s1.index=sp.valueInt AND s2.index=sp.valueInt-1 And sp.index=1;";
pushIntInstruction =
        "UPDATE Stack AS s, Globals AS sp, Globals AS ip, Instructions AS instr " +
            "SET sp.valueInt = sp.valueInt+1, s.valueInt = instr.argint1 " +
            "WHERE s.index=sp.valueInt+1 AND sp.index=1 And " +
            "      ip.index=2 AND instr.instrNo=ip.valueInt;";
jumpNegInstruction =
            "UPDATE stack AS s, Instructions AS instr, Globals AS sp, Globals as ip " +
            "SET ip.valueInt = instr.argInt1-1 "+
            "WHERE s.index=sp.valueInt AND sp.index=1 " +
            "      AND instr.instrNo = ip.valueInt AND ip.index = 2" +
            "      AND s.valueInt < 0";
```

The add instruction updates the stack at the second location from the top index directly, and decrements the stack pointer sp. The push instruction pushes the literal (argint1) stored as part of the instruction and increments the stack pointer sp. The jumpNeg instruction updates the instruction pointer. Notice that the instruction pointer is set to one less than the value in the instruction. As can be seen in the in-

terpreter, this is accounted for as the instruction pointer is always incremented by one after the instruction has been executed.

In the prepared version (appendix B), the only change is that the instruction queries are made as prepared statements.

In the Integrated version (appendix C), we have modified the queries to include updating the instruction pointer as part of the query itself. However, it is not possible to update two rows in the same table using different formulas. As it is rarely the case that the stack pointer and instruction pointer are updated the same way, it is necessary to split the Globals table into two tables, a StackTop table and an InstrPointer table, both with a single row and a single column. The add instruction in this case becomes:

```
addInstruction = con.prepareStatement(
        "UPDATE stack s1, stack s2, StackTop sp, InstrPointer AS ip " +
        "SET s2.valueInt = s1.valueInt+s2.valueInt "+
        ",    sp.valueInt = sp.valueInt-1 " +
        ",    ip.valueInt = ip.valueInt+1 " +
        "WHERE s1.index=sp.valueInt AND s2.index=sp.valueInt-1");
```

The jumpNeg instruction is slightly more complicated, as we need to assign a value based on a condition. Either the next instruction is found by incrementing the instruction pointer or it is taken from the jumpneg instruction itself (argint1). Fortunately, Access has a switch expression that enables exactly this:

```
jumpNegInstruction = con.prepareStatement(
        "UPDATE stack AS s, Instructions AS instr, StackTop as sp, InstrPointer as ip " +
        "SET ip.valueInt = switch( s.valueInt < 0, instr.argInt1, true, ip.valueInt+1)"+
        "WHERE s.index=sp.valueInt AND instr.instrNo = ip.valueInt" );
```

If the first expression in the switch is true, the second expression is the result, else if the third expression is true, the fourth expression is returned. The switch expression is not standard SQL. Other versions of SQL typically offer a similar expression however, e.g. named choice or case.

A.2 Stored procedures

The next set of experiments was to use stored procedures, and to write the interpreter itself as a stored procedure. Access does not provide stored procedures, so we have to investigate a different database, and we have chosen SQL Server 2005 Express, which at the time of writing is free of charge.

Unlike in plain SQL, the Transact-SQL language in SQL Server does not allow an update-statement to update more than one table. This actually makes the instructions easier to code and easier to read. The full solution is shown in Appendix D. Below is shown the procedure for push:

```
PROCEDURE dbo.InstrPush
AS
    UPDATE StackPointer
    SET value = value + 1;

    UPDATE Stack
    SET valueInt = Instructions.argint1
    FROM Instructions, InstructionPointer, StackPointer
    WHERE [index] = StackPointer.value AND Instructions.instrNo = InstructionPointer.value;

    UPDATE InstructionPointer
    SET value = value + 1
    RETURN
```

The interpreter itself was coded as a StoredProcedure:

```
PROCEDURE dbo.RunInterpreter
AS
    SET NOCOUNT ON;
    exec ResetVM;
    DECLARE @instrCount as int;
    SET @instrCount = 0;
    DECLARE @instrType AS VARCHAR(30)
    WHILE 0<1 BEGIN /* True did not work, so I wrote 0<1 */
        BEGIN TRANSACTION;
        SET @instrCount = @instrCount +1;
        SET @instrType = (SELECT instrType
                          FROM Instructions, InstructionPointer
                          WHERE Instructions.instrNo = InstructionPointer.value);
        IF (@instrType='exit') BEGIN
            COMMIT TRANSACTION;
            BREAK;
        END;
        EXEC ('Instr'+@instrType);
        COMMIT TRANSACTION;
    END
    RETURN @instrCount;
```

We exploit a naming convention to call the stored procedures representing the instructions of the virtual machine, in that the name of the stored procedure is named 'instr' concatenated with the name of the instruction type. This will also make it easy to extend the virtual machine with new instructions without actually changing the RunInterpreter procedure.

A.3 DLinq experiment

The final experiment is to use the C# 3.0 DLinq library. Here the idea is to define Entity classes for each of the tables, and write the interpreter in C#, but in such a manner that one synchronizes the database after the execution of each instruction. The code is shown in Appendix E. In the previous tables, StackPointer and InstructionPointer have had but a single row and column. However, when one need to map between rows and objects one must equip each row with a key to preserve object id. In this experiment, the two tables have been extended with an extra column named processID, in anticipation of future use.

The class for representing Stack is shown below. The other tables follow similar structure:

```
[Table(Name="Stack")]
public class StackElement{
   private int index;

   [Column(Id=true)]
   public int Index {
      get { return index; }
      set { index = value; }
   }

   private int valueInt;
   [Column]
   public int ValueInt {
      get { return valueInt; }
      set { valueInt = value; }
   }
}
```

The interpreter itself is this time written in C#, and looks like this:

```
public static int Run(bool submit){
      InitVM();
      SP.Value = -1;
      IP.Value = 1;
      Instruction current;
      int count = 0;
      while(true){
         current = Program[IP.Value-1];
         count++;
         switch (current.InstrType){
            case "exit": return count;
            case "push": {
               SP.Value++;
               Stack[SP.Value].ValueInt = current.ArgInt1;
               IP.Value = IP.Value + 1;
               break;
            }
            case "add":{
               Stack[SP.Value-1].ValueInt += Stack[SP.Value].ValueInt;
               SP.Value--;
               IP.Value++;
               break;
            }
            case "jumpneg":{
               IP.Value = (Stack[SP.Value].ValueInt<0 ? current.ArgInt1 : IP.Value + 1);
               break;
            }
         }
         db.SubmitChanges();
      }
      return count;
   }
}
```

As can be seen, one uses the entity classes without concern for their persistent representation. However, after each instruction, the call db.SubmitChanges() synchronizes the database with the entities.

3. Timing Results

Three experiments were carried out. The timing results are shown below, as number of instructions carried out *per second*.

	Plain	Prepared	Integrated	SProc	DLinq
Average	154	353	454	628	22547
Std. Dev	0,6	3,4	2,9	10,7	142,9
Rel speed	1,0	2,3	2,9	4,1	146,5
Loop	1000	1000	1000	5000	200000

The first three measurements were carried out in Windows safe mode with a command prompt, which provides minimal background noise. However, SQL Server 2005 Express was not able to run in simple Safe mode. Due to the higher speed of the later strategies, the last two used longer loops. The number of loops performed to obtain the results is listed above.

The relative speeds of the three approaches are not surprising. It is well known that prepared statements are faster than plain statements, and it is well known that each roundtrip to the database incurs extra overhead. In the integrated approach, there is only one roundtrip per instruction, whereas the prepared approach uses two – one to execute the instruction, and one to increment the instruction pointer.

It is unfortunately hard to tell if the stored procedure approach is better than the integrated approach, as it is implemented on top of a different database.

However, the performance of DLinq is significantly faster, 146 times the plain solution, but more noticeable, 35 times faster than the stored procedure approach. Both experiments use the same database.

The result is surprising, as object relational mappings are normally considered slow. It is possible to expect the actual SQL used by DLinq. This shows that precise updates are generated. Thus, rather than updating the stack with a 'where stack.index = stackpointer.value', the actual values are used 'where stack.index = 1'. All joins are eliminated using this technique, which might be part of the explanation.

In addition, the program itself is cached in a list in the DLinq example. However, it is worth noticing that it is not possible to do this optimization in the other cases, as no data-structures beside tables are provided in Transact-SQL. One might have considered to replicate the program table in an in-memory temporary table in connection with the stored procedure approach, but that does not get you anywhere, as the program table is already cached by the underlying infrastructure.

Despite the relative speed of the DLinq approach, this has to be one of the slowest interpreters ever built.

4. Outlook

The simple programs discussed above merely hint what a persistent virtual machine might look like. It is far from the idea of a workflow engine. Below is a list of things that need to be done:

- Add data structures (in the form of tables) for process queues, to enable scheduling and synchronization.
- Extend the instruction set. Besides providing the obviously needed arithmetic instructions, full relational operators etc. it is worth to consider making it an extensible interpreter. In particular, it would be useful to be able to write new instructions as stored procedures. This will enable a stored procedure to be a single instruction and give back some of the lost speed.

However, for now, we are able to execute close to 22000 instructions a second. If we assume an office with 2000 employees, each completing one workflow task every 5 minutes, we need to do approximately seven task transactions per second, which gives us room for using approximately 3100 instructions per task. With this margin, we find it reasonable to continue.

The next step is to define a concurrency model for the virtual machine. It must support waiting for a specified duration, and for the occurrence of external events. In addition, it must enable more traditional internal process synchronization. With the relative feasibility of the persistent virtual machine in place, designing the high-level language now needs attention as well.

5. References

We are not aware of anyone who has attempted to build a virtual machine in this way. If you come across similar ideas, I should be quite happy to hear about it – email me at kasper@itu.dk.

In a broader sense, work on process migration is relevant, as it concerns serialization of the execution state for transmission over a network. Process migration differs from our approach in there being specific snapshot points in the program execution in which the process can be serialized for transmission. One can say that our virtual machine is serialized after each instruction.

The issue of supporting workflow processes by being able to persist running threads is also addressed in other work, e.g. [1]. However, most work on persistency addresses transparent data persistence.

1. Florian Matthes Joachim W. Schmidt. Persistent Threads, Proceedings of the Twentieth Conference on Very Large Databases, 1994, Santiago, Chile.

A Plain source code

```
/// Stackmachine, Access database, plain statements
import java.sql.*;
import static java.lang.System.*;
class Access_Plain{

    public static void main(String... args){
        try{
            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
            StackMachine sm = new StackMachine();
            sm.run();
        }catch(ClassNotFoundException uups){
            out.println("Database driver not found");
        }
    }
  static class StackMachine{
    /********* Start of class *****************/

    final private static String dbUrl = "jdbc:odbc:StackInterpreter";

    Connection con;
    String clear1,clear2;
    String getInstruction;
    String addInstruction;
    String jumpNegInstruction;
    String pushIntInstruction;
    String incrIP;
    String incrStack;
    String decrStack;

    String getStackTop;


    public void run(){
        try{
        Statement stmt = con.createStatement();
            //out.println("Starting stack machine");
            stmt.execute(clear1);
            stmt.execute(clear2);
            long start = currentTimeMillis();
            int count = interpret( stmt );
            long end = currentTimeMillis();
            out.print("Instructions executed: " + count);
            out.print(": Execution time: " + ( end-start ));
            out.println(":Instr pr. second: " + ( count*1000/(end-start)) );
            con.close();
        }catch(SQLException uups){
            out.println("Error: " + uups);
            uups.printStackTrace();
        }
    }


    public StackMachine() {
        try{
            con = DriverManager.getConnection(dbUrl);
            clear1 = "UPDATE stack s, Globals g SET g.valueInt=0 , s.valueInt = 0 ";
            clear2 = "UPDATE Globals g SET g.valueInt=-1 where g.index = 1";
            getInstruction =
          "Select instrType from Instructions,Globals " +
```

```
                "WHERE Instructions.instrNo= Globals.valueInt and Globals.index=2";
            addInstruction =
          "UPDATE stack AS s1, stack AS s2, Globals AS sp " +
                "SET s2.valueInt = s1.valueInt+s2.valueInt "+
                ",    sp.valueInt = sp.valueInt-1 " +
                "WHERE s1.index=sp.valueInt And s2.index=sp.valueInt-1 And sp.index=1;";
            pushIntInstruction =
          "UPDATE Stack AS s, Globals AS SP, Globals AS IP, Instructions AS I " +
                "SET SP.valueInt = SP.valueInt+1, s.valueInt = I.argint1 " +
                "WHERE s.index=SP.valueInt+1 And SP.index=1 And " +
                "      IP.index=2 And I.instrNo=IP.valueInt;";
            jumpNegInstruction =
                "UPDATE stack AS s, Instructions AS I, Globals AS sp, Globals as IP " +
                "SET IP.valueInt = I.argInt1-1 "+
                "WHERE s.index=sp.valueInt And sp.index=1 " +
                "      AND I.instrNo = IP.valueInt AND IP.index = 2" +
                "      AND s.valueInt < 0";
            incrIP =
                "UPDATE Globals SET valueInt = valueInt + 1 WHERE index=2";
            incrStack =
                "UPDATE Globals SET valueInt = valueInt + 1 WHERE index=1";
            decrStack =
                "UPDATE Globals SET valueInt = valueInt - 1 WHERE index=1";
            getStackTop =
                "Select Stack.valueInt AS v, Globals.valueInt AS sp from Stack,Globals " +
                "WHERE Stack.index= Globals.valueInt and Globals.index=1";
        }catch(SQLException uups){
            out.println("Could not connect to database\n" + uups);
            uups.printStackTrace();
            exit(1);
        }catch(Exception uups){
            out.println("Something else went wrong: " + uups);
            exit(1);
        }
    }

    public int interpret(Statement stmt) throws SQLException{
        int count = 0;
        while(true){
            count++;
            ResultSet instr = stmt.executeQuery(getInstruction);
            instr.next();
            String opcode = instr.getString("instrType");
            if (opcode.equals("push") ){
                stmt.execute(pushIntInstruction);
                stmt.execute(incrIP);
            }else if (opcode.equals("add") ){
                stmt.execute(addInstruction);
                stmt.execute(incrIP);
            }else if (opcode.equals("jumpNeg") ){
                stmt.execute(jumpNegInstruction);
                stmt.execute(incrIP);
            }else if (opcode.equals("exit") ){
                break;
            }else{
                out.println("Unknown opcode: " + opcode);
            }
        }
        return count;
    }
```

```
    } // end Stack machine

}
```

B Prepared source code

```
/// Stackmachine, Access database, prepared statements
import java.sql.*;
import static java.lang.System.*;
class Access_Prepared {

    public static void main(String... args){
        try{
            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
            StackMachine sm = new StackMachine();
            sm.run();
        }catch(ClassNotFoundException uups){
            out.println("Database driver not found");
        }
    }
  static class StackMachine{
    /********* Start of class *****************/

    final private static String dbUrl = "jdbc:odbc:StackInterpreter";

    Connection con;
    PreparedStatement clear1,clear2;
    PreparedStatement getInstruction;
    PreparedStatement addInstruction;
    PreparedStatement jumpNegInstruction;
    PreparedStatement pushIntInstruction;
    PreparedStatement incrIP;
    PreparedStatement incrStack;
    PreparedStatement decrStack;

    PreparedStatement getStackTop;

    public StackMachine() {
        try{
            con = DriverManager.getConnection(dbUrl);
            clear1 = con.prepareStatement(
                "UPDATE stack s, Globals g SET g.valueInt=0 , s.valueInt = 0 " );
            clear2 = con.prepareStatement(
                "UPDATE Globals g SET g.valueInt=-1 where g.index = 1" );
            getInstruction = con.prepareStatement(
                "Select * from Instructions,Globals " +
                "WHERE Instructions.instrNo= Globals.valueInt and Globals.index=2");
            addInstruction = con.prepareStatement(
                "UPDATE stack AS s1, stack AS s2, Globals AS sp " +
                "SET s2.valueInt = s1.valueInt+s2.valueInt "+
                ",    sp.valueInt = sp.valueInt-1 " +
                "WHERE s1.index=sp.valueInt And s2.index=sp.valueInt-1 And sp.index=1;");
            pushIntInstruction = con.prepareStatement(
                "UPDATE Stack AS s, Globals AS SP, Globals AS IP, Instructions AS I " +
                "SET SP.valueInt = SP.valueInt+1, s.valueInt = I.argint1 " +
                "WHERE s.index=SP.valueInt+1 And SP.index=1 And " +
                "      IP.index=2 And I.instrNo=IP.valueInt");
            jumpNegInstruction = con.prepareStatement(
                "UPDATE stack AS s, Instructions AS I, Globals AS sp, Globals as IP " +
                "SET IP.valueInt = I.argInt1-1 "+
                "WHERE s.index=sp.valueInt And sp.index=1 " +
                "      AND I.instrNo = IP.valueInt AND IP.index = 2" +
                "      AND s.valueInt < 0");
            incrIP = con.prepareStatement(
                "UPDATE Globals SET valueInt = valueInt + 1 WHERE index=2");
```

```
            incrStack = con.prepareStatement(
                "UPDATE Globals SET valueInt = valueInt + 1 WHERE index=1");
            decrStack = con.prepareStatement(
                "UPDATE Globals SET valueInt = valueInt - 1 WHERE index=1");
            getStackTop = con.prepareStatement(
                "Select Stack.valueInt AS v, Globals.valueInt AS sp from Stack,Globals " +
                "WHERE Stack.index= Globals.valueInt and Globals.index=1");
        }catch(SQLException uups){
            out.println("Could not connect to database\n" + uups);
            uups.printStackTrace();
            exit(1);
        }catch(Exception uups){
            out.println("Something else went wrong: " + uups);
            exit(1);
        }
    }


    public void run(){
        try{
            clear1.executeBatch();
            clear2.executeBatch();
            long start = currentTimeMillis();
            int count = interpret();
            long end = currentTimeMillis();
            out.print("No. Instructions executed: " + count );
            out.print(":Execution time: " + ( end-start ));
            out.println(":Instr pr. second: " + ( count*1000/(end-start)) );
            con.close();
        }catch(SQLException uups){
            out.println("Error: " + uups);
            uups.printStackTrace();
        }
    }

    public int interpret() throws SQLException{
        int count = 0;
        while(true){
            count++;
            ResultSet instr = getInstruction.executeQuery();
            instr.next();
            String opcode = instr.getString("instrType");
            if (opcode.equals("push") ){
                pushIntInstruction.executeUpdate();
                incrIP.executeUpdate();
            }else if (opcode.equals("add") ){
                addInstruction.executeUpdate();
                incrIP.executeUpdate();
            }else if (opcode.equals("jumpNeg") ){
                jumpNegInstruction.executeUpdate();
                incrIP.executeUpdate();
            }else if (opcode.equals("exit") ){
                break;
            }else{
                out.println("Unknown opcode: " + opcode);
            }
        }
        return count;
    }
} // end Stack machine
```

```
}
```

C Integrated source code

```
/// Stackmachine, Access database, prepared statements, integrated ip update
import java.sql.*;
import static java.lang.System.*;
class Access_Prep_Integrated {

    public static void main(String... args){
        try{
            Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
            StackMachine sm = new StackMachine();
            sm.run();
        }catch(ClassNotFoundException uups){
            out.println("Database driver not found");
        }
    }
  static class StackMachine{
    /********* Start of class *****************/

    final private static String dbUrl = "jdbc:odbc:StackInterpreter";

    Connection con;
    PreparedStatement clear1;
    PreparedStatement getInstruction;
    PreparedStatement addInstruction;
    PreparedStatement jumpNegInstruction;
    PreparedStatement pushIntInstruction;


    public void run(){
        try{
            clear1.executeUpdate();
            long start = currentTimeMillis();
            int count = interpret();
            long end = currentTimeMillis();
            out.print("No. Instructions executed: " + count );
            out.print(":Execution time: " + ( end-start ));
            out.println(":Instr pr. second: " + ( count*1000/(end-start)) );
            con.close();
        }catch(SQLException uups){
            out.println("Error: " + uups);
            uups.printStackTrace();
        }
    }

    public StackMachine() {
        try{
            con = DriverManager.getConnection(dbUrl);
        clear1 = con.prepareStatement(
                "UPDATE stack s, InstrPointer ip, StackTop sp " +
          "SET s.valueInt = 0, ip.valueInt=0 , sp.valueInt = -1  " );
            getInstruction = con.prepareStatement(
                "Select instrType from Instructions,InstrPointer ip " +
                "WHERE Instructions.instrNo= ip.valueInt");
        addInstruction = con.prepareStatement(
                "UPDATE stack s1, stack s2, StackTop sp, InstrPointer AS ip " +
                "SET s2.valueInt = s1.valueInt+s2.valueInt "+
                ",    sp.valueInt = sp.valueInt-1 " +
          ",    ip.valueInt = ip.valueInt+1 " +
                "WHERE s1.index=sp.valueInt And s2.index=sp.valueInt-1");
        pushIntInstruction = con.prepareStatement(
```

```
                "UPDATE Stack AS s, StackTop as SP, InstrPointer as ip, Instructions AS I " +
                "SET SP.valueInt = SP.valueInt+1, s.valueInt = I.argint1 " +
          ",    ip.valueInt = ip.valueInt+1 " +
                "WHERE s.index=SP.valueInt+1 And I.instrNo=IP.valueInt");
        jumpNegInstruction = con.prepareStatement(
                "UPDATE stack AS s, Instructions AS I, StackTop as sp, InstrPointer as IP " +
                "SET IP.valueInt = switch( s.valueInt < 0, I.argInt1, true, Ip.valueInt+1)"+
                "WHERE s.index=sp.valueInt AND I.instrNo = IP.valueInt" );
        }catch(SQLException uups){
            out.println("Could not connect to database\n" + uups);
            uups.printStackTrace();
            exit(1);
        }catch(Exception uups){
            out.println("Something else went wrong: " + uups);
            exit(1);
        }
    }

    public int interpret() throws SQLException{
        int count = 0;
        while(true){
            count++;
            ResultSet instr = getInstruction.executeQuery();
            instr.next();
            String opcode = instr.getString("instrType");
            if (opcode.equals("push") ){
                pushIntInstruction.executeUpdate();
            }else if (opcode.equals("add") ){
                addInstruction.executeUpdate();
            }else if (opcode.equals("jumpNeg") ){
                jumpNegInstruction.executeUpdate();
            }else if (opcode.equals("exit") ){
                break;
            }else{
                out.println("Unknown opcode: " + opcode);
            }
        }
        return count;
    }
  } // end Stack machine
}
```

D Stored Procedure Solution

```
PROCEDURE dbo.RunInterpreter
AS
    SET NOCOUNT ON;
    exec ResetVM;
    DECLARE @instrCount as int;
    SET @instrCount = 0;
    DECLARE @instrType AS VARCHAR(30)
    WHILE 0<1 BEGIN /* True did not work, so I wrote 0<1 */
        BEGIN TRANSACTION;
        SET @instrCount = @instrCount +1;
        SET @instrType = (SELECT instrType
                          FROM Instructions, InstructionPointer
                          WHERE Instructions.instrNo = InstructionPointer.value);
        IF (@instrType='exit') BEGIN
            COMMIT TRANSACTION;
            BREAK;
        END;
        EXEC ('Instr'+@instrType);
        COMMIT TRANSACTION;
    END
    RETURN @instrCount;

ROCEDURE dbo.ResetVM
AS
    UPDATE InstructionPointer
  SET InstructionPointer.value=1;
  UPDATE StackPointer
  SET StackPointer.value=-1
  RETURN
PROCEDURE dbo.InstrPush
AS
    UPDATE StackPointer
    SET value = value + 1;

    UPDATE Stack
    SET valueInt = Instructions.argint1
    FROM Instructions, InstructionPointer, StackPointer
    WHERE [index] = StackPointer.value AND Instructions.instrNo = InstructionPointer.value;

    UPDATE InstructionPointer
    SET value = value + 1
    RETURN

PROCEDURE dbo.InstrJumpNeg
AS
    UPDATE InstructionPointer
    SET value =
        CASE WHEN Stack.valueInt < 0 THEN Instructions.argint1
            ELSE InstructionPointer.value + 1
        END
    FROM Stack, StackPointer, Instructions
    WHERE Stack.[index] = StackPointer.value AND  Instructions.instrNo = InstructionPointer.value
    RETURN
PROCEDURE dbo.InstrAdd
AS
    DECLARE @topvalue int;
    SET @topvalue = (SELECT Stack.ValueInt
                     FROM Stack, StackPointer
                     WHERE Stack.[index] = StackPointer.value);
```

```
        UPDATE StackPointer
        SET value = value -1;
        UPDATE Stack
        SET ValueInt = ValueInt + @topvalue
        FROM Stack, StackPointer
        WHERE [index] = StackPointer.value;

        UPDATE InstructionPointer
        SET value = value + 1
RETURN
```

E DLinq Solution

```
namespace POPTest {
    public class VM{
        public static DataContext db;

        public static void InitVM(string connectionString){
            db = new DataContext(connectionString);

            IP = db.GetTable<InstructionPointer>().Head();
            SP = db.GetTable<StackPointer>().Head();

            Table<Instruction> prog = db.GetTable<Instruction>();
            Program = new List<Instruction>(prog.OrderBy(instr => instr.InstrNo));

            Table<StackElement> st = db.GetTable<StackElement>();
            Stack = new List<StackElement>(st.OrderBy(sta => sta.Index));

        }
        public static InstructionPointer IP;
        public static StackPointer SP;
        public static List<StackElement> Stack;
        public static List<Instruction> Program;

        public static long Run(string connectionString, bool submit){
            InitVM(connectionString);
            SP.Value = -1;
            IP.Value = 1;
            Instruction current;
            long count = 0;
            while(count<100000000){ // to execute only few ~ use count <10
                current = Program[IP.Value-1];
                count++;
                switch (current.InstrType){
                    case "exit": return count;
                    case "push": {
                        SP.Value++;
                        Stack[SP.Value].ValueInt = current.ArgInt1;
                        IP.Value = IP.Value + 1;
                        break;
                    }
                    case "add":{
                        Stack[SP.Value-1].ValueInt += Stack[SP.Value].ValueInt;
                        SP.Value--;
                        IP.Value++;
                        break;
                    }
                    case "jumpneg":{
```

```
                IP.Value = (Stack[SP.Value].ValueInt<0 ? current.ArgInt1 : IP.Value + 1);
                break;
            }
            default:{
                Console.Out.WriteLine("Unknown Instruction");
                return count;
            }
        }
        db.SubmitChanges();
    }
    return count;
  }
}


[Table(Name="InstructionPointer")]
public class InstructionPointer{
  private int _processID;
  [Column(Id=true)]
  public int ProcessID {
    get { return _processID; }
    set { _processID = value; }
  }

  private int id;
  [Column(Name="value",Id=true)]
  public int Value {
    get { return id; }
    set { id = value; }
  }


}
[Table(Name="Instructions")]
public class Instruction{
  private int _instrNo;
  [Column(Id=true)]
  public int InstrNo {
    get { return _instrNo; }
    set { _instrNo = value; }
  }

  private string _instrType;
  [Column]
  public string InstrType {
    get { return _instrType.Trim(); }
    set { _instrType = value.Trim(); }
  }

  private int _argInt1;
  [Column]
  public int ArgInt1 {
    get { return _argInt1; }
    set { _argInt1 = value; }
  }

}
[Table(Name="Stack")]
public class StackElement{
  private int index;
```

```
        [Column(Id=true)]
        public int Index {
            get { return index; }
            set { index = value; }
        }

        private int valueInt;
        [Column]
        public int ValueInt {
            get { return valueInt; }
            set { valueInt = value; }
        }

    }
    [Table(Name="StackPointer")]
    public class StackPointer{
        private int _processID;
        [Column(Id=true)]
        public int ProcessID {
            get { return _processID; }
            set { _processID = value; }
        }

        private int _value;
        [Column]
        public int Value {
            get { return _value; }
            set { _value = value; }
        }

    }

    public static class ExtensionMethods{
        public static T Head<T>(this IEnumerable<T> en){
            var enu = en.GetEnumerator();
            enu.MoveNext();
            return enu.Current;
        }
    }
}
```