INTERNATIONAL JOURNAL OF ADVANCED RESEARCH IN ENGINEERING AND TECHNOLOGY (IJARET)

ISSN 0976 - 6480 (Print)
ISSN 0976 - 6499 (Online)
Volume 4, Issue 5, July – August 2013, pp. 153-163
© IAEME: www.iaeme.com/ijaret.asp
Journal Impact Factor (2013): 5.8376 (Calculated by GISI)
www.jifactor.com

IJARET
©IAEME

OVERLAPPING OPTIMIZATION WITH PARSING THROUGH METAGRAMMARS

Nadera Beevi S[1]*, Dr. Chitraprasad D[2] and Dr.Vinod Chandra S.S[3]

[1]Associate professor, Department of MCA, TKM College of Engineering, Kollam, Kerala, India.
[2]Professor and Head, Department of CSE, TKM College of Engineering, Kollam, Kerala, India.
[3]Director, Computer Centre, University of Kerala,Trivandrum, India.

ABSTRACT

This paper describes techniques for improving the performance of meta framework developed by combining C++ and Java language segments through reducing the number of bytecodes generated. Augmented versions of existing languages can be developed by combining good properties of those languages. It increases the flexibility of programmers in using language constructs of those languages. The framework identifies and parses source code with C++ and Java language statements using metagrammar developed and create a unified AST for the hybrid source code. Bytecodes are generated for AST and interpreted. The performance of Bytecodes can be improved through optimization techniques associated with metagrammars, like constant propagation which identifies constant values for variables and propagate it to the place where the variable occurs and replace it with corresponding value. Function inlining and exception optimization greatly improves the execution time performance of Bytecodes. Optimization through metagrammars eliminates rigorous analysis of bytecodes to identify hot spots and optimize them.

Keywords: C++, Java, metaframework, Lex, Yacc, bytecodes, optimization.

1. INTRODUCTION

Compilers are used to translate a source program to target program. Meta compilers are available that can compile more than one language. Multiple keyword constructs used in a single program is came to be known as a Meta language. It is a program transformation methodology for developing efficient programs by combining good properties of different languages. Meta language can be considered as a standard technology for software maintenance and evolution [1]. But functional equivalence between source program and target program should be maintained. There is a large research project called *Meta*, developed whose main aim is to increment and unify the syntax and semantics of existing languages [2]. We have developed a framework from Java and C++ which is designed in such a way that class developed in one language can be used in other language and vice versa. It provides better flexibility for programmers in using various language constructs.

The meta framework developed generates Abstract Syntax Trees (AST) for hybrid source code. Bytecodes are generated from AST and interpreted. Data flow optimizations such as constant propagation, constant folding, algebraic simplification can improve execution time performance of bytecodes. Constant propagation identifies constant value of a variable and propagate them to the place where variable occurs and replace it with corresponding value. It helps in simplification of algebraic expressions by evaluating expressions at compile time. It greatly reduces execution time evaluations especially inside a loop. While generating bytecodes about 40% of instructions are to load operands from local variable area of the stack to the operand area of the stack. These instructions are followed by bytecodes accessing operands from stack top. Constant propagation through meta grammars optimizes local loads by generating bytecodes with native operands. Data Flow optimizations at source level is applied while generating Abstract Syntax Trees. When source program is translated to low level intermediate representation source level information about control structures such as conditional branches and loops are lost and it is difficult to create target high level language program close to source program. Loops are the important sources of optimizations and information about loops can be saved through source level optimizations. Function calls and returns also occur frequently in the source program. The speed of execution of method calls can be greatly improved through method inlining. Static methods as well as dynamic methods converted to static ones can be inlined. Embedded code generators associated with metagrammar rules converts dynamic method to static methods and perform method inlining while generating Abstract Syntax trees. Exception handling mechanism is another important source of optimization. The performance of bytecodes can be greatly improved by bypassing unnecessary exception checks.

Our goal is to develop an optimized meta framework for a program containing keyword constructs from C++ and Java. The input to our system is a precise description of source and target language. The syntactic and semantic information about component languages of meta are mixed to make meta parser easy to understand and maintain. The syntactic elements of the language are considered as strings and grouped into syntactic structures. Context free grammars are used to describe the structure of meta language. Syntax and Semantics of two languages are clearly defined by Recursive Functions on Context Free (CFRF) languages through which syntax check of the language can be done [3], [4]. Semantic analysis of the language can be done through attribute grammars. The system uses syntax equations resembling BNF into which output intermediate language commands are inserted. Embedded code generators associated with meta grammar rules optimizes and develops an intermediate AST for source program. The AST is scanned to produce bytecodes which are interpreted. The following section gives a brief review of the related study.

2. RELATED WORKS

Program transformation enhances the implementation of software systems and applications in multiple languages. The performance of program transformation can be improved by generating optimized object codes. There are many related works performing source to source program transformations mainly concentrating on Execution Preserving Language Transformation (EPLT) [5],[6],[7]. But only partial translation is carried out in their work. Active involvement of the user is needed before producing final output. Many developers consider source to source program transformation as expensive, time consuming and therefore infeasible. So one solution proposed is to wrap it and embed it in new application without changing the language rather than redeveloping [8], [9]. There have been a few attempts of implementing language transformation using meta frameworks[10],[11],[12]. The basic intention is to provide flexibility in using data structures for programmers. Any change requiring design decisions is not recognized or reported. Reeuwijk talks about a template based meta compiler for generating source code of any programming language [12].

In this work a template language **Tm** was developed. It accepts data structure definitions and source code template as input and produce target code of a particular programming language as output.

Algorithms to achieve high level global data flow optimizations such as constant propagation are extensively elaborated in[13],[14],[15]. The program is analyzed to create a program flow graph to propagate constants from assignment to usage site. Even though running time is improved in each of the succeeding works, the analysis is done on intermediate code and the construction of flow graph is time consuming.

A declarative optimization technique for data flow optimization is present in path logic programming[16]. Regular expression provides necessary information to propagate constant values to the usage site from the assignment region. The repeated analysis of the graph is required for propagating constants dynamically which is not required in our approach. A framework for improving the performance of Java programs using Java class file attributes is presented in [17]. It uses Soot byte code optimization framework to optimizes the byte codes. The program is analyzed to find information about various class attributes which is used to convey profile information and perform array optimization.

All the above algorithms perform data flow optimizations on low level intermediate form. Constant propagation using System Dependence Graph (SDG) is suitable for high level optimizations where a data flow interpreter is used to propagate constant values from the assignment node to the usage site[18]. But the construction of SDG is time consuming. A novel approach to eliminate redundant array checks and provide accurate error messages about faulty array references using array resizing is developed by Thi Viet Nga Nguyen et al [19]. Additional array checks are inserted with minimum slow down in performance. The properties of above algorithms is used to improve the performance of meta framework through meta grammar rules in our work.

3. IMPLEMENTATION

Program transformation frameworks promote pure code migrations and help to reengineer legacy systems to object oriented platforms. The input of the system is a meta program containing C++ and Java language constructs. Source program expressed as sequence of characters is translated into a representation for use in the meta framework. The source language is specified using a context free grammar, meta grammar and a *lexing* and a parsing method can be used to perform the translation. Figure 1 shows the Compilation model developed.

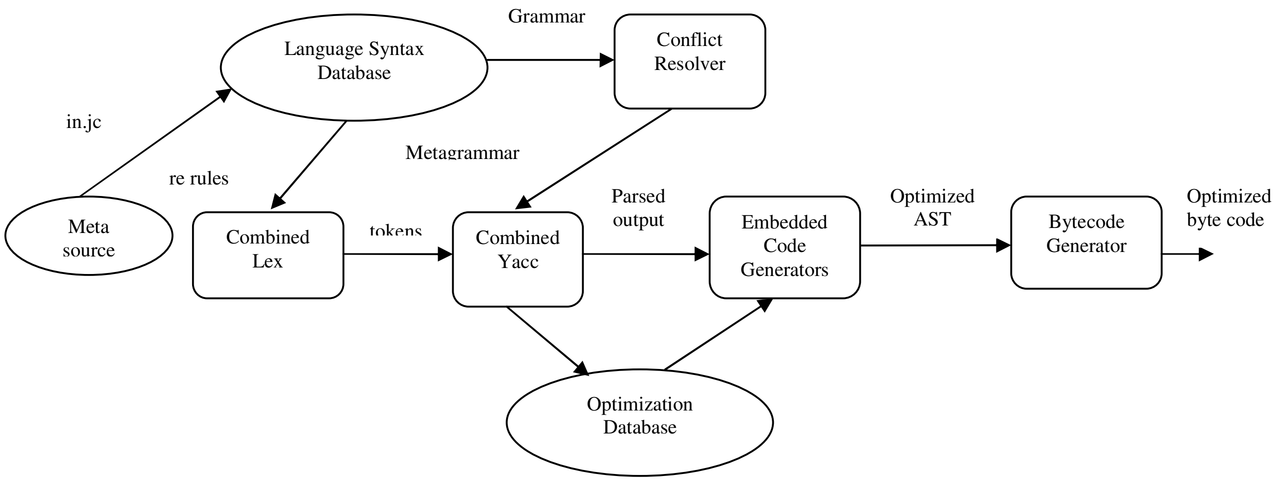

Fig. 1. Metaplatform Structure

The lexical patterns in a meta program are identified using *lex* tool. The meta program is preprocessed and the output is used by *Yacc* to generate tokens from lex file. *Yacc* is a tool that generates syntax analyzer for the source code. The meta-grammar of our system is a combination of C++ and Java grammar. Various types of conflicts arise when we try to combine grammars of different source languages. Around 100 shift reduce or reduce-reduce conflicts identified while parsing the input. Slight changes in production rule disambiguate the grammar and parse input correctly. The definition section of *Yacc* file is concerned with token definitions of component languages. The rule section provides grammar rules resembling BNF with code defining clauses added. These embedded code generators generates Abstract Syntax Trees for input program. The speed of execution of hybrid source code can be improved though optimization techniques associated with these semantic actions. Data flow optimizations such as constant propagation, constant folding, unreachable code elimination etc can be performed through grammar rules developed for the framework. Dynamic rewriting rules are required for the propagation of information since the context information required is not available at the usage site. Also if any intervening definition occurs for the variable, propagation becomes invalid. Static methods and exception checks are other sources of optimization. Various optimization techniques developed through meta grammar for the framework are discussed.

3.1. Constant Propagation

Constant propagation identifies values of variables that are constants and propagate it to the place where the variables are used. It is used for the optimized generation of byte codes. Unnecessary local loads can be eliminated by generating bytecodes with immediate operands identified through constant propagation. The working of algorithm to discover constants in sequential statements, conditional branches eliminating unreachable codes and functions are discussed here.

3.1.1 Sequential Code

The constants defined through #define statements is changed by preprocessor to final constants. In preprocessing stage itself the program is scanned and all final constants are replaced by corresponding value. The replacement of variable with value should done carefully because constant propagation should not be done if an intervening definition occurs for the variable. The code generators associated with metagrammar for variable assignment creates a variable table containing the variable and a pointer to the symbol table to access the value. If a constant assignment is encountered a flag field in the table is set to true. While creating AST for the expression, the variable table is scanned and if constant flag is set for the variable, constant propagation is done for it. If an intervening variable assignment occurs, the flag is set to false. The metagrammar implementing constant propagation strategy in straight line code is shown below.

```
assignment: name assignment_operator assignment expression
{
 if (literal_flag)
        {
        strcpy(const,yytext);
         insert (const,name) into cons_table);
        set const_flag=true;
         literal_flag=false;
        }
}

assignment_expression: additive|multiplicative|conditional
```

```
additive: multiplicative|additive PLUSmultiplicative|additive   MINUS multiplicative

{
        strcpy(id,yytext);
       Const_prop_strategy=(id∈const_table&&(const_flag))?assign(id,value):assign(id,var);
  }

multiplicative: unary
unary: PLUS unary|MINUS unary|name|literal
{
if (literal) literal_flag=true);
}
```

Constant propagation generates bytecodes with native operands. It reduces the number of resultant bytecodes. Constant pool look up is also minimized, for we are generating instructions with immediate operands. An example follows demonstrate the fact.

Example:

```
Class E1
public int  operation (int a,int b)
{
int c=2;
int t1=a+b*c;
return t1;
}
```

Corresponding Constant Pool entries are

1. Methodref class=2 signature =4
2. classRef name =3
3. UTF8Text "java/E1"
4. Name/type name = 5 type =6
5. UTF8text "operation"
6. UTF8text ("II;")I
7. Const Integer 2

Bytecodes for the above method

- Iload_0
- Iload_1
- ldc 7
- Imul
- Iadd
- ireturn

String propagation saves two constant pool look ups. Consider the method to find the length of a string

Constant pool entries are

1. Classrefname=3
2. String constant value =10
3. UTF8text "java/E2"
4. Methodref class=5 sig =7
5. Classref name=6

6. UTF8text “java/lang/String”
7. Name/type name=8 type=9
8. UTF8text “length”
9. UTF8text “()”I
10. UTF8text “welcome”

Corresponding bytecodes

- ldc 2
- invokevirtual 4
- pop
- return

Strings converted into character array eliminates the need of creating constant pool entries

- aload_0
- invokevirtual #index //char[] java.lang.String.toCharArray()
- astore content

Corresponding bytecodes for string length

- aload content
- arraylength
- return

3.1.2 Conditional Branches and loops

Control flow structures like conditional branches and loops bifurcate and iterate the data flow based on condition evaluated. Unreachable code can be eliminated by identifying paths that are not executable through constant propagation and constant folding. Thus it saves execution time computations especially inside a loop.

For example

```
c = 10;                        c = 10;
b = 12;                        b = 12;
 if ( a>(c+b))                  if (a>22)
{                              {
a = a+b;                       a = a+12;
c = 10;                        c = 10;
}                               }
else                                 else
{                              {
a = b;                         a = 12;
c = 12 ;                       c  = 12;
}                               }
```

The values of **c** and **b** can be propagated through meta grammar associated with expressions and constant folding is done while creating Abstract Syntax trees. Information of constants is propagated outside the loop if they are assigned with same values in all execution paths taken for the condition evaluated. So the value of **c** cannot be propagated. The metagrammar associated with unreachable code elimination is as follows:

cond_stat: |if LPARA expression RPARA stat|
if LPARA expressionRPARAstatELSEstat
{ survive_strategy=constant_fold_epression?if_code_AST:else_code _AST }

3.1.3 Constant Propagation along functions

Constant Propagation along method boundaries helps to generate efficient object code. Constant propagation using meta grammar uses intraprocedural flow information to propagate constants. Consider the following program code.

```
main()
{
    a = 10;
     if(a==10)
     fun1(a,x);
     else
    fun2(12)
 }

fun1(c,d)     fun2( y)
{ }              { }
```

Constant propagation strategy implemented through the meta grammar of assignment statement propagates constant to variable '**a**'. Unreachable code elimination procedures associated with the rules for conditional structures and expression evaluation makes an intra flow sensitive analysis of main function and propagate constants for **fun1** only. The meta grammar associated with constant propagation for function is as follows.

```
method_invocation :   name LPARA argument_list_opt RPARA
                      |primary DOT IDENTIFIER LPARA argument_list_opt RPARA
                      |SUPER DOT IDENTIFIER LPARA argument_list_opt RPARA
argument_list_opt  :argument_list
argument_list      :expression   |argument_list COMMA expression
```

The code to propagate literal constants to formal arguments while reducing the above grammar for method call is as follows.

```
for each class in the program
      for each function call in the class
         for each actual argument a of the function call
           {    let f be the corresponding formal argument
                 if a is a literal constant
               {   insert (a,f) into cons_table.
                   propagate a to f in function definition. }
               if a is a variable and element of constant table
               {  propagate constant value from the table to f      }  }
```

3.2 Method Inlining

The speed of execution of method call can be increased through method inlining. Static methods in a class and object constructors can be inlined. In our system while creating Abstract Syntax Trees the derived objects are statically determined by the system and virtual calls are replaced with direct procedure calls. If the method is overridden in a single subclass, the virtual call is statically linked, reverting virtual function to a non virtual one. Otherwise a series of runtime class tests is done for the expected receiver classes replacing virtual method call with a direct procedure call to the corresponding class found. The following algorithm identifies the receiver classes of dynamically dispatched methods.

```
for each class C in the program
{    for each method m in the class
          {          insert m () to the method list of the class
                     if it is virtual   set the vflag for the method
                     If it is in the method list of any other class P
                     Add P to the class list of method m       }  }
```

Class hierarchy analysis helps to identify the methods visible to a particular class. Union of the method sets is done to find methods visible in derived classes. Intersection of the method set is performed to identify class set where the method is visible. A base pointer table is created for each base class used in the program. It contains information about the derived classes of particular base class. The following meta grammar identifies virtual method call and generates a dynamic replacement strategy that invokes method of the corresponding receiver class.

```
Cpp_point_type : MULT name
type_decl: type cpp_point_type { if type ∈ base_class, base_obj=name;}
Rep_strategy = assignment : |left_name EQ AMPERSAND name ,left_name ∈ base_obj and
                                        name ∈ der-obj ? der_call:base_call
```

Since the information about the derived object is not available at the usage site dynamic modification of the base pointer is needed.

3.3 Exception Check Elimination

Another source of optimization lies in exception handling. Exceptions prevents one of the most common programming errors, array bound violations. It results in unexpected outputs and failures. Programming languages like Java allow access of array elements within the declared range. Array bound check helps to determine whether all array references are within the declared range. It cause JVM to execute a compare instruction which check the access is within the array limit. It consists of two bound checks per dimension. It increases the size of executable files, compilation and execution time. Execution time of program can be greatly improved by eliminating unnecessary exception checks.

Type checks are of two types. Fully redundant and partially redundant [20]. Fully redundant checks can be removed if it can be proved that they never fails at compile time. Partially redundant checks include checks whose number can be reduced. For example, array bound checks inside the loop can be replaced with another check outside the loop. A new kind of error checking is introduced in metaframeworks where metagrammar verifies whether array references in the program are within the declared range. The array bound check can be reduced by comparing array length with range of index value used in the loop. The metagrammar for array declaration identifies array length and value is passed to the point where loops are parsed and array is used. Range of the index variable of loop is compared with array length and if the array index inside the loop passes range test , no exception check is done for the array. Otherwise the loop is executed with a safe version containing exception handling code.

Array statements are implemented by creating a new node in the AST. When the program is parsed itself, all the array assignments are identified and a pointer to the AST of the array assignment nodes are saved. A dedicated Array Table is used for this purpose which contains a pointer to the array using node and an index to the symbol table to access the details of array including dimensions. An array_Check_Required flag is added in the node to determine whether array check is required at run time or not. If array check is not required exception handling code is not inserted at run time. And also information about faulty references consisting of line number, array name, faulty index is also displayed. Unnecessary exception check removal implicitly provides dead code elimination also.

Exception Check Elimination Algorithm

```
1. Array Declaration
        identifier EQUALS  NEW type LBRACK LITERAL RBRACK {
                1.1 SYM_TAB[new_entry]->name= $1->identifier
                1.2 SYM_TAB[new_entry]->type=$4->type
                1.3 SYM_TAB[new_entry]->dimension=$6
                1.4 ARR_TAB[new_entry]->name= $1->identifier
                1.5 ARR_TAB[new_entry]->null_check_required =false.,.
2. Array Assignment
        variable LBRACK  LITERAL RBRACK  EQUALS LITERAL
         {

                2.1 entry: search ($1->identifier, ARR_TAB)
                2.2 ARR_TAB[entry]->index:search(SYM_TAB,$1->identifier)
                2.3 for each dimension di of ARR_TAB[entry]->index ->dimensions

                    if  (($3<=di) && ($3>=li))
                    {
                       array_check_required=false;

                        temp=  AST_NODE(ARR_TAB[entry])
                    }
                    else
                    {
                    array_check_required=true;
                    Display (line number,arrayname,faulty_index)
                    }
   }
```

Array bound check implicitly includes null pointer check also. The length of the array is stored at a small offset from the array address. Since length is needed for bound check, null check of array address is automatically done while accessing . While implementing array bound check, null check is also implemented through metagrammar during parsing itself. No bound check instructions are added, if both checks are safe. If null check is not safe, corresponding instructions are added in the bytecode.

cmp r, r+offset
jge outofbounderror
Array Bound Check instructions

4. CONCLUSION

We have described the concepts behind optimization of meta framework developed using meta grammar. The optimization methods implemented while parsing hybrid source code eliminates rigorous analysis of bytecodes for optimization. The benefits identified are (1) Optimization using meta grammar retains source level information about conditional branches and loops. It makes construction of object code that executes exactly same as source program easier. (2) It improves the execution performance of byte codes eliminating unnecessary loads and stores instructions. (3) Constant propagation at compile time saves a lot of run time computations especially inside a loop. (4) Constant propagation along control structures easily identifies unreachable codes and eliminates it. (5) Metagrammar simplifies intraprocedural flow analysis to propagate constants along procedure boundaries. The implementation of optimization techniques through lex and yacc accelerate the execution performance of hybrid source code containing C++ and Java constructs using the metagrammar developed. The data and control flow of the meta language is modified to obtain functionally equivalent output.

REFERENCES

1. Balzer R.M, Goldman N.M, and Wile D.S, "On the Transformational Implementation Approach to Programming", In Proceedings of the 2nd International Conference on Software Engineering, 337-344, 1986.
2. Holst W, "Meta: Extending and Unifying languages", OOPSALA'04 , 331-344, 2004.
3. Dong Y, "Recursive functions of Context free Languages(I) – The definitions of CFPRF and CFRG", Science in China, Series F, 45(1): 25-39, 2002.
4. Dong, Y, "Recursive functions of Context free Languages(II) – Validity of CFPRF and CFRG definitions", Science in China, Series F, 45(2):1-21, 2002.
5. Malabarba S, Devanbhu P.T and Stearns A, "MoHca Java: A tool for C++ to Java Conversion support", In Proceedings of International Conference on Software Engineering, ICSE, 650-653, 1999.
6. Buddrus F, Schode J, "Cappuccino - a C++ to java translator", In Proceedings of the 1998 ACM Symposium on Applied Computing, 660-665, 1998.
7. Laffra G," C2J: a C++ to Java translator", Novosoft, 2001.
8. Sneed H.M ," Wrapping legacy COBOL programs behind an XML interface", In Proceedings of WCRE ,IEEE Computer Society Press,2001, 189-197.
9. Sneed H.M, "Migrating from COBOL to Java", In Proceedings of International Conference on Software Maintenance (ICSM), IEEE, 2010, 1-7.
10. Galler B, Perlis A.J, "A Proposal for definition in ALGOL, ACM Communication", 1967, 10, 204-219.
11. Vinod Chandra S.S , Achuthsankar, S. Nair, "A MetaL for C and Pascal", SIGCSE Bulletin, ACM , 2007, 39, 4, 87-91.
12. Van Reeuwijk C, "Rapid and Robust Compiler Construction Using Template-Based Metacompilation", Springer Verlag Berlin Heidelberg, 2003,247-261.
13. Kildall G.A, "A unified approach to global program optimization", In Conference recordings of the first ACM Symposium on Principles of Programming Languages, 194-206, 1973 .
14. Wegman M, Zadeck F, "Constant Propagation with conditional branches", ACM Transactions on Programming Languages and Systems, 1991, pp. 181-210.
15. Wegbreit B, "Property extraction in well-founded property sets", IEEE Transactions on Software Engineering, 1975, pp. 270-285.
16. Drape S, de Moor,O and Sittampalam G, 'Transforming the .NET intermediate language using path logic programming in: C", Kirchner,editor, Proceedings of the Fourth ACM SIGPLAN Conference on Principles and Practise of Declarative Programming, ACM, Pittsburg, USA, 2002, pp.133-144.
17. Raja vallee-Rai, Phong Co, Ettenne Gagnon, Laurie Hendren, Patric Lam, Vijay Sundaresan, "Soot- a Java bytecode optimization framework", In Proceedings of the CASCON'99, 1999, pp.214-224.

18. Binkley D, "Interprocedural Constant Propagation using dependence graph and a data-flow mode", in:mFritzson, editor, International Conference on Compiler Construction (CC'94), Lecture Notes in Computer Science(LNCS), 1994, pp.374-388.
19. Thi Viet Nga Nguyen, Francois Irigoin, "Effiecent and Effective array bound checking", Transactions on Programming Languages and Systems (TOPLAS), ACM, 2005,27, pp. 527-570.
20. Thomas Wurthinger, Christian Wimmer, Hanspeter Mossenbock, "Array bound check elimination in the context of deoptimization", Science of Computer Programming, ELSEVIER, 2009, 74, pp.279-295.
21. Santosh Kumar Pani and Priya Arundhati, "An Effective Methodology for Slicing C++ Programs", International Journal of Computer Engineering & Technology (IJCET), Volume 1, Issue 1, 2010, pp. 57 - 71, ISSN Print: 0976 – 6367, ISSN Online: 0976 – 6375.
22. Prof. S.A.Ubale and Dr. S.S. Apte, "Study and Implementation of Code Access Security with .NET Framework for Windows Operating System", International Journal of Computer Engineering & Technology (IJCET), Volume 3, Issue 3, 2012, pp. 426 - 434, ISSN Print: 0976 – 6367, ISSN Online: 0976 – 6375.
23. Pratibha S. Yalagi and Dr. Sulabha S. Apte, "Exploiting Parallelism for a Java Code with an Efficient Parallelization Technique", International Journal of Computer Engineering & Technology (IJCET), Volume 3, Issue 3, 2012, pp. 484 - 489, ISSN Print: 0976 – 6367, ISSN Online: 0976 – 6375.