A Statistical Analysis of the Benefits of Partial Evaluation on C Function Calls

By

LIANG, Linda L.

A Thesis
Submitted to the Faculty of Graduate Studies
in Partial Fulfillment of the Requirements
for the Degree of

MASTER OF SCIENCE

Department of Computer Science
The University of Manitoba
Winnipeg, Manitoba

THE UNIVERSITY OF MANITOBA

FACULTY OF GRADUATE STUDIES

*****

COPYRIGHT PERMISSION PAGE

A STATISTICAL ANALYSIS OF THE BENEFITS OF PARTIAL EVALUATION ON C FUNCTION CALLS

BY

LINDA L. LIANG

A Thesis/Practicum submitted to the Faculty of Graduate Studies of The University of Manitoba in partial fulfillment of the requirements of the degree of MASTER OF SCIENCE

Abstract

Partial evaluation is a source-to-source program transformation technique which substitutes the constant or known part of the input at compilation time to get an updated program. To help make decisions as to what kind of partial evaluation to implement, statistics have been collected on the use of translation-time constant arguments in C language source code. Source code from three important projects coded in C was analyzed: the X-Windows library, the LINUX operating system, and the Gnu C compiler "gcc." These projects also present typical modern uses of the C language. The frequency of constant arguments in argument lists can have important implication on the worth of certain code improvement strategies. Specifically, compiler writers and partial-evaluator designers can benefit from these statistics if they are planning to do any of the following: replacing calls of functions with all constant arguments by function results, unfolding function calls, or specializing functions for specific values of actual arguments. The statistics collected show that these partial evaluation techniques can yield significant performance improvement for some projects, at reasonable costs in memory usage.

Acknowledgements

First, I wish to express my deep thanks to my supervisor professor Daniel J. Salomon. He introduced me to the field of programming language and has been giving me much valuable advice and guidance throughout the preparation of the thesis. At the same time, I wish to express my thanks to all members of my Advisory Committee for their reading this thesis and giving many good suggestions.

Also, my thanks are expressed to the Department of Computer Science and professor Daniel J. Salomon for the financial support through a NSERC grant.

Finally, I wish to express my sincere appreciation to my family for the full understanding and the constant encouragement.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Introduction

This thesis modifies a C recognizer to collect statistics on function calls in C, and analyzes how much benefit can be obtained from applying partial evaluation to C code. Therefore, it is also an evaluation of the potential benefit provided by partial evaluation to a new language called Safer_C, which is a modern descendant of the C language.

1.2 Partial Evaluation

Partial evaluation is a source-to-source program transformation technique which substitutes the constant or known part of the input at compilation time to get an updated version of the program. The principal goal of partial evaluation is to increase execution speed. It can improve the efficiency of programs by exploiting known information about the input of a

program, performing some computation at compilation time, and generating a transformed program which can run faster than the original one.

Although the principle of partial evaluation is simple, its implications are surprisingly complex, especially when applied to imperative languages. The analysis of programming features for partial evaluation is an active area in the field of programming language design and implementation.

1.3 A Survey of the Literature on Partial Evaluation

This thesis surveys some recent results in the area of partial evaluation, especially issues dealing with the partial evaluation of imperative languages. Given a general program and part of its input, partial evaluation deals with specializing the program with respect to this known information.

Using the notation of Consel & Danvy[2], consider a program p and its input i, and say that somehow we can split i into a static (i.e. known) part s and a dynamic (i.e. unknown) part d. The literal constants used in a program can also be considered as part of s. Given a specializing function S, we can specialize p with respect to s:

$$S\ (p,\langle s,\text{-}\rangle) = Ps$$

By definition, running the residual program Ps must yield the same result as the general program would yield, provided both terminate:

$$\text{run}\ p\ \langle\ s,d\ \rangle = \text{run}\ Ps\ \langle\ \varnothing,d\ \rangle$$

The objective of partial evaluation is to produce a residual Ps that runs

faster than p.

The major areas in my research topic and papers in those areas are:

(1) Memoizing[10]. This is an early work in the field of partial evaluation. It proposes the concept of "Memo" functions which we can use to store the values of some functions which may be used frequently later. In that way the program can be speeded up.

(2) Partial evaluation [2,5,7,8,9,12]. This area consists of the concept of partial evaluation, and its various areas of applications which include compiling and compiler generation, numerical computation, and hard real-time systems.

(3) Partial evaluation for imperative languages [2, 3,4,5,7,11]. This area presents some techniques for and applications of partial evaluation of imperative languages such as Pascal, C, and FORTRAN.

1.4 Function Specialization

Function specialization is a technique of partial evaluation to generate a specialized version of a function for each value of constant parameters.

For example, for function call Work(2,A), we can create a specialized version of that function called Work_2(A) for function calls where the value of the first parameter is 2. In most cases, this can increase the speed of the execution of the program.

1.5 Collecting Statistical Data on Function Calls in C

This thesis project collects statistical data on typical function calls in C code. The information collected includes:

(1) How many functions have all parameters constant? In this case, functions with no side effects can be replaced by their respective results.

(2) How many functions have 1/n, 2/n, 3/n, ..., n-1/n constant parameters (where n is the total number of the parameters in the function call)?

(3) How many specialized versions would be obtained?

To determine if a function can be specialized, collect data on:

(a) How often the same constant parameter has the same value?

(b) How often is this constant parameter the only constant value?

1.6 Methodology

An existing C recognizer is modified to interpret the C code and to retrieve the related information. The information is output to a file. The data on the file is analyzed to get the statistical results. .

The source code for following software were chosen for analysis because of importance, wide use, and availability.

(1) The X Windows system

(2) The gcc compiler

(3) The Linux operating system

1.7 Organization of the Thesis

This thesis is organized as follows. Chapter 2 is a survey of partial evaluation and its applications. The principles of partial evaluation are presented. The basic strategies of partial evaluation are introduced. Several partial evaluation applications are also presented. In chapter 3, the information to be collected from the input C code is demonstrated. How this information is related to the study of partial evaluation of imperative languages is also stated. In chapter 4, the data collection method is presented. Our methodology is to interpret the input C code from the output of the parser, analyze the retrieved information, and obtain statistical data on the input C code. Some important implementation decisions are also discussed. In chapter 5, the statistical results of three selected widely used software - X Windows, the Linux system, and the gcc compiler are presented. Discussion of these results is also stated. In chapter 6, a conclusion is presented. It was found that polyvariant specialization of function calls with constant parameters is feasible.

Chapter 2

Survey of Current Issues in Partial Evaluation

2.1 Introduction

This chapter discusses the concepts of partial evaluation and contains a survey of current results in this field.

2.2 What Is Partial Evaluation?

Given a general program and part of its input, partial evaluation deals with specializing this program with respect to this known information.

Using the notation of Consel & Danvy[2], consider a program p and its

input i, and suppose that somehow we can split i into a static (i.e. known) part s and a dynamic (i.e. unknown) part d. Literal constants used in the program can also be considered to be part of the known input. Given a specializing function S, we can specialize p with respect to s

$$S\ (p, \langle s,- \rangle) = P_S$$

By definition, running the residual program P_S must yield the same result as the general program would yield, provided both terminate:

$$\text{run}\ \ p\langle s,d\rangle = \text{run}\ \ Ps\ \langle \varnothing,d\rangle$$

The objective of partial evaluation is to produce a residual program P_S that runs faster than p.

2.3 How to Perform Partial Evaluation

2.3.1 Two Basic Strategies of Partial Evaluation

The purpose of partial evaluation is to specialize a program with respect to some known parts of input. The two basic strategies are folding and unfolding .

2.3.1.1 Folding Constant Expressions

The technique of folding constant expressions consists of propagating constant values, executing any constant expressions that result, and replacing the expressions by the results.

2.3.1.2 Unfolding & Unrolling

Unfolding can be applied to control structures and function calls.

If the control flow of a program sequence can be determined at compile time, partial evaluation will unfold iterative loops, and reduce the conditional structures to one of their branches.

Here is an example of unrolling control structures.

Consider the following statements.

```
If  ( i > 10 )  then  j += 2;
                else  j += 4;
```

Also, assume we have i = 15 at compile time.

Applying partial evaluation to the above code segment, one of the selective branch is eliminated. We obtain a simpler statement: j += 4;

If a few of the parameters are known at compile time, partial evaluation will inline functions by unfolding calls, and produce specialized functions by residualizing calls.

For example, let's consider the following function definition.

```
int lamda(int x, int y)
{
    int result;
    result = (x + y ) / 2;
    return result;
}
```

For function call lamda(4,b), we can unfold the function and get a residual function lamda_4(int z).

```
int  lamda_4(int z)
{
    int result;
    result =  2 + z/2;
    return result;
}
```

2.3.2 Online vs Offline Partial Evaluation

Partial evaluators are divided into two classes: online and offline (Consel & Danvy[2]).

An online partial-evaluator is a non-standard interpreter. The treatment of each expression is determined on the fly. Online partial-evaluators are very accurate but actually they have considerable overhead.

For example, we have the following code segment in C.

```
void main(argc, argv)
{
    int  known, unknown;
    int  result1, result2;
```

```
    result1 = sqr(known);
    result2 = ( known + unknown ) / 2;
}
```

An online partial evaluator will determine if an expression can be evaluated according to the status of variables in it. The compile-time variables are declared at the beginning. Here is the code segment with the initial declaration.

```
void main(argc, argv)
{
    compile_time int  known;
    run_time int      unknown;
    int  result1, result2;
    result1 = sqr(known);
    result2 = ( known + unknown ) / 2;
}
```

Having the initial compile_time variables declaration, the partial evaluator goes through every expression in the program and determines its status according to the declaration. Variable *result1* is classified as a compile_time variable since *known* is a compile_time variable. Variable *result2* is a run_time variable since it calls a compile_time variable *known* and also a run_time variable *unknown*. We can see an online partial evaluator is accurate but it is also slow.

Offline partial evaluators are structured with a preprocessing phase PE and a processing phase PE. The preprocessing phase PE usually includes *binding-time analysis*. Given the binding-time signature of a source program (i.e., which part of the input is static and which part is dynamic), the binding-time analyzer propagates this information through the source program, determining for each expression whether it can be evaluated at compile-time or whether it must be evaluated at run-time. An offline partial-evaluator is less accurate than an online partial-evaluator since binding-time analysis is approximate. An example of an offline partial evaluator for FORTRAN 77 will be given in a later section.

Online and offline partial-evaluation can be combined to get the best results. When the accurate binding-time property of an expression is known, offline partial-evaluation should be used. Otherwise, the evaluation of this expression is postponed until specialization-time; at this time concrete values are available.

2.3.3 The tradeoff of Partial Evaluation

Using partial evaluation, we can get a transformed program (which is called the residual program) which can be run faster than the original one. But usually the residual program is bigger than the original one due to loop unrolling and function unfolding. That could be the tradeoff: taking more space for the programs and data can produce faster execution while taking

less space can lead to slower computation.

Nevertheless it can happen that a program can be made both smaller and faster. A well known example of this as given by Consel & Danvy[2] is the program xphoon written by Poskanzer and Leres. That's the best case.

2.4 A Summary of the Issues in the Field of Partial Evaluation

This section will give a survey of the history and issues in the field of partial evaluation.

2.4.1 Memoizing

This is an early work in partial evaluation. In his paper [10], Donald Michie proposed a facility to convert functions into "memo functions." By doing this, a program can improve the speed of the evaluation of numerical functions a great deal.

2.4.1.1 The "Rule Part" and the "Rote Part"

In his paper, Michie presented an efficient way to evaluate mathematical functions, in which a program can avoid needless tests and redundant evaluation by "learning from experience."

For each function, there is a "rule part" which is the computational procedure and a "rote part" which is a "look up" table. That is, the "rule part" is the operation by which the function is evaluated, the "rote part" is the

table in which the value of the function is stored or other look-up medium is stored after it has been computed. The "rote part" is made up of results from previously computed function evaluations. Here is an example of the "rule part" and the "rote part" of the function *fact*.

The rule part of *fact* is:

```
function fact n;
  if n < 0 or if not (n isinteger) then undef
  else
    if n = 0 then 1 else n*fact(n-1)
  end
```

Suppose now the first call of the function is *fact*(4). At first the rote part is invoked, but no entry for 4 is found. So the rule part is invoked. The value of n is not equal to 0. The answer is 4*fact(3). We now set out to evaluate fact(3). We check the rote again. There is no entry for 3 either. Then we enter the rule and find 3*fact(2). We check the rote and again do not get an entry for 2. Now we have to come back to the rule and repeat the process until we encounter fact(0). We can find the value of it by rule, which is 1. We can now evaluate fact(1) = 1 * fact(0), and hence fact(2) = 2*fact(1), and hence fact(3) = 3*fact(2), and fact(4) = 4*fact(3). Each of these evaluation adds a new entry to the rote part. Then at the end of the recursion, the rote part looks like this:

Argument	Value
4	24
3	6
2	2
1	1
0	1

According to Michie, for each function,

(1) The apparatus of evaluation associated with any given function consists of a "rule part" and a "rote part";

(2) The evaluation in the computer should on each given occasion proceed either by rule, or by rote, or by a blend of the two, solely as dictated by the expediency of the moment;

(3) The rule versus rote decisions can be handled by the machine behind the scenes;

(4) various kinds of interaction are permitted to occur between the rule part and the rote part.

2.4.1.2 Self-Improvement of This Scheme

There could be a lot of improved methods of this scheme. Michie proposed an improvement which we can call the "move-to-front" algorithm. In

this algorithm, rarely-occurring problems will move towards the bottom of the rote table and frequently occurring problems move towards the top.

Any other searching algorithms such as hashing, most-common-first, LRU could also be used to improve to this scheme.

2.4.1.3 Application of This Scheme

This scheme can be applied to many digital computations to improve evaluation speed, especially for the recursively defined functions, because it is obvious that we can reuse the previous calculation results in the recursive functions.

2.4.1.4 The Relationship Between Partial Evaluation and "Memo Functions"

The "Memo Function" scheme was developed in the early stage of partial evaluation research. It gives the idea of efficient computation by the elimination of redundant calculation, storing the most frequently used data values which are going to be used in the evaluation.

2.4.2 Partial Evaluation of Imperative Languages

This section will present some techniques for the partial evaluation of imperative languages. Some applications are also introduced.

2.4.2.1 Characteristics of Partial Evaluation for Imperative Languages

A lot of research has been done on the partial evaluation of functional languages. In actual fact most of the "real world programs" are written in imperative languages such as C, COBOL, FORTRAN, etc. So the partial evaluation of imperative languages has received more attention recently.

The partial evaluation of imperative programs is more difficult than that of functional programs. Because of the lack of referential transparency, the program transformation phase must take into account the notion of state. The replication of side-effects is the main concern.

2.4.2.2 Techniques for Partial Evaluation of Imperative languages

Various techniques for the partial evaluation of imperative languages are presented in Consel & Danvy [2], Meyer [3], Nirkhe & Pugh [11]. This section will summarize those techniques.

2.4.2.2.1 Dynamic Notation

Partial evaluation could get stuck in two ways: either by unfolding infinitely many function calls or by creating infinitely many specialized functions. To avoid the non-termination problem, Meyer[3] simply assumes the program will terminate.

According to the definition of partial evaluation, a partial evaluator has

to decide whether the current statement (form, clause, etc) has to be executed or not. Execution of the statement will change the state of the memory, and non-execution will add the statement into the residual program.

To provide the proper information to the partial evaluator, there are two strategies (as we mentioned in the previous section): "static annotations" and "dynamic annotations".

Static annotations are inserted into the text of the program in a separate phase before partial evaluation. They are either computed by hand or using abstract interpretation. All parts of the program with possibly unknown data have to be annotated. The term "annotation" here means making a note of the status of each variable. According to Meyer, this technique is appropriate for functional languages because variables in functional languages cannot change their values. In Meyer's opinion, the proper technique for imperative language is to make dynamic annotations. The user has to annotate only the declarations of the variables which should be treated as unknown data. These annotations give us information about the relation between variables and their states, i.e., known or unknown. The partial evaluator has to look up the state of a variable when it encounters this variable. The state of a variable might change from known to unknown and vice versa, if there is an assignment to that variable. With dynamic annotations the partial evaluator will decide dynamically whether the current statement has to be executed or suspended, depending on the states

of the involved variables. Meyer also gives a formal description for this method [3].

2.4.2.2.2 Specialization of Subroutines

Specialization is an important technique for partial evaluation. In this section we will describe how this technique can be adapted to imperative languages.

In the following discussion, we discard the cases in which global variables are used in the bodies of the subroutines (we will discuss those cases in section 2.4.3.3). We will present the techniques for function calls and procedure calls .

Specializing Function Invocations

For function invocations, we need to handle three cases.

(1) All the Values of the Actual Arguments Are Known At Compile-Time

When a partial-evaluator encounters a function invocation, it has to check the values of the actual arguments. If all of them are known, the partial evaluator will bind the formal parameters to the values of the actual arguments. With these variables and the local variables, the body of the function is evaluated. If we get residual statements (for example, the statement includes another function invocation which can't be evaluated

totally at this time), we have to construct a residual function containing these residual statements as the body, and output the residual function. Otherwise the function can be evaluated totally. We do not get a residual call and there is no residual function. Just replace the function invocation by the result of the evaluation.

(2) Not All the Values of the Actual Arguments Are Known at Compile-Time

If not all the values of the actual arguments are known at compile-time, the formal parameters should be suspended. We will get a residual function.

For each function call, a list is evaluated. The list will contain the values of actual arguments (for those known actual argument values), and the formal parameters (for those parameters without the matching known arguments). For example, consider the function definition A(X,Y), for a compile-time known input X = 2, the list is (2,Y).

These lists are stored in a table together with the name of the original function and the name of the specialized version of that function. When there is a function invocation, the partial evaluator will look up the table for a pattern with the name of the function and a list matching the actual arguments. If such a pattern has been found, the copy of the corresponding specialized version of the function is used for the construction of the residual function call. The specialized functions are the functions of the residual program.

If the termination condition of the function call depends on the variables in an unknown state, this method for specializing function calls might not terminate. To avoid this problem, Meyer [3] proposed the solution that the user can label the critical function calls with a symbol. When such a call is evaluated, only the actual arguments are evaluated and composed into a residual function call. The original function is copied into the residual program.

Specializing Procedure Calls

There are three cases in specializing procedure calls.

(1) Call-by-value Parameters

For those procedure calls where parameters are passed by the mode call-by-value, parameters are handled in the same way as in the case of function calls. They will appear in the parameter list when the corresponding actual argument is unknown at the specialized time.

(2) Call-by-result Parameters

Call-by-result parameters appear in the parameter list only when the state of the variable for that parameter is unknown after the execution of the body of the procedure. The state of the actual argument is irrelevant.

(3) Call-by-reference Parameters

In this case, the formal parameters and the actual arguments denote the same memory location. There is only one state for both.

In the case of parameters transmitted by location, if the value of the actual argument is unknown, it has to appear in the list of the parameters for the residual procedure invocation and the formal parameter has to be in the list of the parameters of the definition of the residual procedure. On the other hand, in the case of parameters transmitted by values, if the value of the formal parameter is unknown after the execution of the procedure the actual argument has to appear in the parameter list in the residual function call and the formal parameter has to be in the parameter list of the definition of the residual procedure.

Global Variables and Side-effects

Global variables and side-effects pose a considerable problem in the partial evaluation of imperative languages. The use of global variables in subroutines may cause problems if a call to these subroutines cannot be totally evaluated.

Here is an example of the problem.

```
void main(argc,argv)
{
  int global;
```

```
    int re1, re2, p1, p2;
    int   work(int x, int y)
    {
        int  result;
        global = 5;
        global += x;
        result = ( x + y) / 2 ;
        return result;
    }
    re1 = work(2,p1);
    re2 = work(2,p2);
}
```

After the execution of the program, the value of the global variable "global" is 9.

Applying partial evaluation to the above code, we have a residual function work_2.

```
int work_2(int z)
{
    int  result;
    global = 7;
    result = 1 + z/2;
    return result;
```

```
}
```

The invocation of the residual functions will be

```
re1 = work_2(p1);
re2 = work_2(p2);
```

After the execution of the program the value of "global" is 7, which is obviously wrong.

Meyer [3] proposes to set the states of all global variables used in the subroutines to "unknown" to solve this problem. An algorithm is presented to compute the set of all global variables occurring in a subroutine or in subroutines reachable from a subroutine. The state of each global variable in the set is set to unknown. Then the techniques for specializing subroutines we described in the previous sections can be applied.

2.4.2.3 Applications of Partial Evaluation of Imperative Languages

Due to its conceptual simplicity and efficiency, partial evaluation has been applied to various areas which include numerical computation [7,13], hard real-time systems [5,11], and compiling and compiler generation [9,12].

In this section, we will illustrate several application areas of the partial evaluation of imperative languages.

2.4.2.3.1 Numerical Computation

Partial evaluation is very effective in cases when some parts of the input change less frequently than others. That's the reason why specializing numerical computation algorithms by partial evaluation can give substantial savings. This section will describe some characteristics of numerically oriented programs and a partial evaluator for a subset of FORTRAN 77 [7,13].

(1) The Characteristic Features of Numerical Problems

A characteristic feature of numerically oriented problems is that most of their numerical computations require dynamic data and therefore cannot be computed during specialization. But in many cases the control flow can be determined at specialization time. For example, for any given matrix size, matrix-multiply performs a fixed set of multiplication's, even though the numerical values of the elements might be unknown at compile-time.

Numerical programs can be divided into two types: data-independent and data-dependent code sequences. A sequence of operations is *data-independent* if the control flow can be determined at compile time. Otherwise it is *data-dependent.* The largest part of numerical programs are usually data-independent. Partial valuation works best for data-independent code because iterative loops can be unfolded and reduced to one of the branches, while the control flow in data-dependent computation which depends on dynamic values cannot be determined by partial evaluation.

(2) A Partial Evaluator for FORTRAN 77

In [7], a partial evaluator for a substantial subset of FORTRAN 77 is presented.

(2.1) The Subset of FORTRAN 77

The language selected for the partial evaluator is a subset of FORTRAN 77, called F77. It includes multidimensional arrays, functions, procedures, and COMMON regions. The statements include assignments, nested conditionals, IF, unconditional jumps, GOTO, procedure calls, CALL, function calls, the RETURN, and the CONTINUE statement, Expressions include constants, identifiers, indexed arrays, arithmetic and relational operators.

(2.2) The Partial Evaluator System

The partial evaluator of Baier, Gluck,Zochling [7] is implemented in FORTRAN 77. The input and output of the partial evaluator are programs written in F77. The partial evaluator is off-line. The source program will be binding-time analyzed before it is specialized. The three phases of this partial evaluator system are:

-- The preprocessing phase translates an F77 source program into an

intermediate language, called CoreF. The binding-time analysis (BTA) annotates all statements(expressions) in the source program as either static or dynamic corresponding to the static/dynamic classification(S/D) of its input. The output of the preprocessing phase is an annotated CoreF program.

-- The specialization phase takes the annotated CoreF program and the static data as input and specializes the program with respect to the static data. This phase is the main part of the partial-evaluator system. It contains an interpreter (INT) for the evaluation of static CoreF statements. The output of the specialization phase is a specialized CoreF program.

-- The postprocessing phase translates a specialized CoreF program into F77.

(2.3) The BTA (Binding-Time Analysis) and the Partial Evaluation of Statements

Binding-time analysis classifies each variable as static or dynamic. The analysis used in the BTA is monovariant, i.e., every statement (expression) can be given only one static/dynamic classification. The BTA is implemented by a fixed-point iteration: an approximate algorithm which iterates over the static-dynamic division until a stable classification is reached.

The output of the BTA is an annotated CoreF program with a notation "static" or "dynamic" attached to every variable.

The specialization phase follows the annotation made by the BTA: it executes static statements, reduces partially static expressions, and specializes dynamic program points.

During the specialization of a dynamic basic block, the partial evaluator runs through the sequence of statements step by step, executing static statements and generating code for dynamic ones. When a dynamic conditional, e.g. an IF, is met, both branches are specialized.

The experimental result of using this partial evaluator for a number of numerically oriented problems shows that the residual program runs 3-4 times faster than the original one.

2.4.2.3.2 Partial Evaluation Applications in Hard Real-Time Systems

This section will present how partial evaluation can be applied to the hard real-time systems. A partial evaluator for the Maruti hard real-time system is also introduced.

(1) Problems with Current Hard Real-Time System

Hard real-time systems are applications where it is catastrophic to violate timing constraints (such as weapons control, medical instrumentation). The hard real-time system requires ensuring predictable timing behavior of HRT applications by taking their timing and resource requirement into account. Problems arise when application programs use

high-level constructs (such as recursion, loops, and dynamic data structures). The execution time and other resource requirements are difficult to estimate at compile time. Current techniques in the HRT operating system cannot handle this problem. Their solution to this problem is to forbid or restrict the use of these high-level language features. That causes inconvenience for the system designers, and the developed low-level programs are hard to adapt for different environments.

(2) The Solution of Partial Evaluation

In their paper [5], Nirkhe and Pugh proposed their solution to this problem which is based on partial evaluation. In their method, a programming language is designed. A program in this language can be partially evaluated which produces a residual program that is guaranteed to terminate. The tight upper bounds on execution time and resource utilization of the residual program can be automatically determined. The technique is to transform the program with high-level language features into a new program which has mostly linear code and an estimatable execution time.

(3) A Partial Evaluator in the Maruti Real-time System

This section describes a partial evaluator of Maruti real-time system. Maruti is a hard-real-time operating system which is based on the

technique of pre-scheduling. In this technique, the timeliness of the application execution is ensured by reserving the required resources prior to run-time.

Application programs in Maruti are written in MPL (Maruti Programming Language). In MPL, a block is a collection of statements that have a separate temporal scope. Since blocks have separate timing constraints and resource requirements, they may be scheduled separately. The temporal and precedence relations among blocks is represented in the form of a graph, which is called a computation graph.

Partial evaluation of a program produces a residual program in the form of a computation graph. In the computation graph, nodes consist of sequential code segments. Nodes are labeled with their execution time and resource requirements. During partial evaluation, compound statements such as conditionals and loops are transformed into ones with restricted use of these constructs. For the resulting statements, the bounds of loops and depth of recursion are known and the tight estimate of execution time of blocks is determinable. The result of partial evaluation are nodes of the computation graph, in which edges represent execution ordering and time constraints. This computation graph is used by the Maruti Scheduler to preschedule the program in a way that guarantees that the program will satisfy all of its timing and resource constraints.

In this method, during partial evaluation, the residual program must

be guaranteed to terminate and have a predictable behavior. If such prediction is not possible, errors are detected and reported as type errors. If in some cases partial evaluation fails to terminate after a time period specified by the user, we abort the compilation and print a message describing the portion of the program apparently causing problems. Normal estimation techniques are used with the resulting program to estimate the resource requirements.

(4) Advantage of Partial Evaluation of HRT Programs

The main benefit of partial evaluation of real-time programs is allowing programmers to write using high-level, reuse-oriented programming styles. Traditional techniques of execution time estimation can be used on residual programs to obtain the time and resource requirements. Also, the residual programs are more efficient than the original ones.

2.4.2.3.3 Applications of Partial Evaluation to Compiler Generation

This section describes another important application of partial evaluation: automatic compiler generation.

(1) Partial Evaluation and Interpreters

Interpreters are easier to write than compilers, while compilers are more efficient. Partial evaluation can yield both those advantages. By the

specialization of an interpreter to a program, the net effect is compilation. If the partial evaluator is self-applicable, an interpreter is automatically transformed into a compiler.

The following diagrams illustrate how they relate.

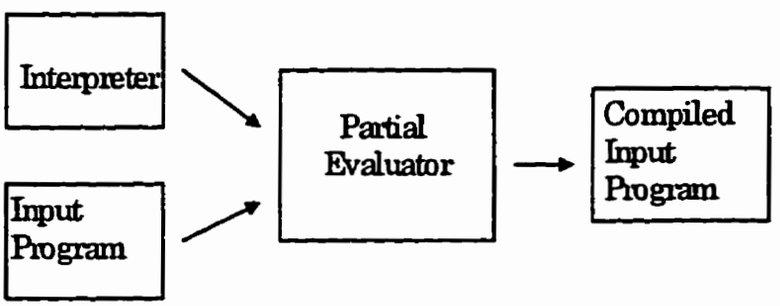

Fig 2.1 Partial Evaluation of An Interpreter Amounts to Compiling

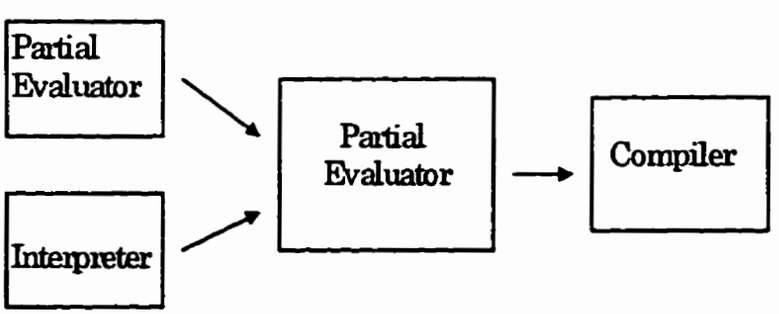

Fig. 2.2 Partially Evaluating A Partial Evaluator Yields A Compiler

In the following subsection, we will give more explanation and a mathematical proof of the above statements.

(2) How Partial Evaluation Works in Compiler Generation

Partial evaluation works in compiler generation in this way: Consider an interpreter for a given language L_s. The specialization of this interpreter to a known source program P_s(written in L_s) already is a target program for P_s , written in the same language L_{int} as the interpreter. So, partial evaluation of an interpreter with respect to a fixed source program amounts to compiling.

The following subsections describe the applications of partial evaluation in compiler generation.

(2.1) Compiling by Partially Evaluating an Interpreter

According to the notation of [12], we denote an interpreter in language L_{int} as **int**, the source language as L_S. programs written in source language as P_S. We define the interpreter **int** as follows.

$$Run\ P_S\ L_S\ \langle d_1, ..., d_n \rangle = Run\ L_{int}\ \mathbf{int}\ \langle P_S, d_1, ..., d_n \rangle \qquad (2.1)$$

for all programs P_S written in language L_S and data $d_1, ..., d_n$.

By this definition an interpreter takes both the program to be interpreted and all input data for the program as its input.

Let the program PE written in language Lp be a partial evaluator for the source language L_S, and let **int** be an interpreter for source language L_S. Let's see what happens if **int** is partially evaluated with respect to a given program P_S written in the language L_S. According to (2.1), we have

$$\begin{aligned} \textit{Run}\ L_S\ P_S\ \langle d_1, ..., d_n \rangle &= \textit{Run}\ L_{int}\ \textbf{int}\ \langle P_S, d_1, ..., d_n \rangle \\ &= \textit{Run}\ L_{int}\ (\ PE\ \langle \textbf{int}, P_S \rangle\ )\ \langle d_1, ..., d_n \rangle \end{aligned} \qquad (2.2)$$

Note that formula (2.2) describes the application of partial evaluation to an interpreter **int**. The result of this is the same as the result of applying the program P_S to data $\langle d_1, ..., d_n \rangle$, so we call this

$$\text{target} = PE\ \langle \textbf{int}, P_S \rangle \qquad (2.3)$$

The program target is a program written in language L_{int} with the same input-output behavior as the source program P_S. In other words we have compiled the source program P_S into a program target written in language L_{int} by partially evaluating the interpreter **int** with respect to the source program P_S.

(2.2) Compiler Generation

To illustrate the application of partial evaluation in compiler generation, we first give a definition of autoprojector.

Definition 2.1: A program *mix* written in language L_p is an autoprojector iff it is a partial evaluator for language L_p. So an autoprojector is a partial evaluator for a language such that the partial evaluator is written in the same language. Assume that an autoprojector *mix* is given. Letting *mix* play the role of the partial evaluator PE in formula (2.3), we have that

$$\text{target} = mix\ \langle \mathbf{int}, P_S \rangle \tag{2.4}$$

We define a computation Com as follows.

$$\text{Com} = mix\ \langle\ mix, \mathbf{int} \rangle \tag{2.5}$$

According to this definition, we have

$$\text{Com}\ P_S = (mix\ \langle\ mix, \mathbf{int} \rangle)\ P_S \tag{2.6}$$

since *mix* is self-applicable, i.e. an autoprojector, according to (2.6), applying partial evaluation to mix itself, we have

$$\text{Com}\ P_S = mix\ \langle \mathbf{int}, P_S \rangle \tag{2.7}$$

Combining formula (2.4) and formula (2.7), we have

$$\text{Com}\ P_S = \text{target} \tag{2.8}$$

Thus Com is a compiler, since given P_S it produces a target program for P_S.

(2.3) Compiler Generator Generation

By the same reasoning a compiler generator ComGen may be obtained by computing:

$$\text{ComGen} = \text{mix}\ \langle \text{mix},\text{mix} \rangle \qquad (2.9)$$

This program ComGen transforms interpreters into compilers, because

$$\text{Com} = \text{ComGen}\ \text{int} \qquad (2.10)$$

The following is a proof of formula (2.10)

$$\text{Com}\ P_S = (\text{mix}\ \langle \text{mix},\text{mix} \rangle, \text{int})\ P_S$$

$$= (\ \text{mix}\ \langle \text{mix},\text{int} \rangle\ )\ P_S$$

$$= \text{mix}\ \langle \text{int}, P_S \rangle \qquad (2.11)$$

Combining formula (2.4) and (2.11), we have

$$\text{Com}\ P_S = \text{target}$$

Chapter 3

Data Collected

3.1 Introduction

A subprogram is an abstract operation defined by the programmer. Subprograms form the basic building blocks out of which most programs are constructed. Explicitly transmitted parameters and results are the major methods of sharing data objects among subprograms. In the C programming language, all subprograms are functions, and the functions are the key parts that do the real work in the C programs. Functions are invoked by other functions and ultimately used by the function main( ) to solve the original problem. This research project collects statistical data on typical function calls in C code to predict how much benefit we can get from applying partial evaluation to C code. This chapter will describe what statistical data is collected, and why we collect these data.

3.2 "Call by Value" Mechanism

Since the purpose of our research is to analyze the frequency of constant parameters in function calls in C code, the following is a brief review of major methods of parameter passing in function invocations.

There are three primary mechanisms for parameter passing: in-out parameters, in-only parameters, and out-only parameters. In C, parameters to functions are always passed "by value", i.e., using the "in-only" mechanism. This means that when an expression is passed as an argument to a function, the expression is evaluated, and it is this value that is passed to the function. The variables passed as arguments to functions are not changed in the calling environment.

Our analysis program collects all the parameters to all function calls in a C program and reports this information to a file called *prog.stat* for further analysis. It also reports whether or not each parameter is a constant. In the case where such a constant parameter is an expression, we output the whole expression to the file *prog.stat* instead of evaluating it, due to the complexity of interpreting C code, at this stage. Although we don't have the evaluated value of this expression, we do report whether or not it itself is a constant value, and keep a record of all these unevaluated expressions for further analysis of possible errors in the resulting statistics.

3.3 Statistical Data Collected in the Project

3.3.1 Function Specialization

As stated in the introduction, the function call is a very important part of C programming. Therefore, when applying partial evaluation to the C language, function specialization and function inlining (unfolding) are important techniques.

3.3.1.1 What is function specialization?

Function specialization is a technique for partial evaluation to generate a specialized version of a function for each value of its constant arguments that is actually used in the program. Let us use an example to illustrate this. Suppose that in a C program *prog.c*, there is a function prototype: *void work(int a, int b)*. Also suppose that there is a function invocation *work(2, c)*. With the principle of partial evaluation, we can evaluate the function with the first parameter being 2 and therefore create a specialized version of that function *work_2(int b)*. This new version, with fewer parameters, will run faster than the original version in most cases. The specialized function may also be smaller than the original version. To apply function specialization, whenever we encounter a function call *work(2,x)* with the first parameter being 2, we can replace it with function call *work_2(x)*. This will usually make execution more efficient. In some interaction with cache and virtual

memory, the specialized function may run slower than the original one due to memory transfers if it is larger than the unspecialized version.

3.3.1.2 How to perform function specialization

As we state in chapter 2, there are 2 different cases in specializing function invocations.

Function invocation with all constant parameters

If all the parameters are known at compile-time, the partial evaluator will evaluate the function with the values of the actual arguments and the local variables used in the function call. Eventually we will get either a residual function (if the function can't be evaluated fully) or a particular result of the evaluation to replace the function call.

Function invocations with some but not all constant parameters

In this case, we will get a residual function with the unknown parameters suspended.

3.3.2 Statistical Data Collected

Since function specialization is an important part of the partial evaluation of imperative languages, to study how much benefit we can obtain when we apply partial evaluation to imperative programs, we have undertaken to

collect statistical data on function calls in standard C code. The following is the information we collected.

3.3.2.1 How many functions have all parameters constant?

The reason to collect this information is that we can determine how many functions can be replaced by an evaluated result (assuming there are no side affects in these functions). Such a change can have a significant effect on execution speed.

3.3.2.2 How many functions have 1/n, 2/n, 3/n, ..., n-1/n constant parameters (where n is the number of the parameters in the function call) ?

The layout of the output generated is the following:

		No. of Constant Args											Total
		0	1	2	3	4	5	6	7	8	9	10	
No. of Arg	0	4											4
	1		4										4
	2	11	7	2									20
	3		5	4									9
	4	9	8		6	1							24
	5		2	10		3							15
	6	3	14	5			2						24
	7					5		1	1				7
	8				2			1					3
	9						1				1		2
	10					1		1			1	1	4
Total		27	40	21	8	10	3	3	1	0	2	1	116

Table 3.1 Distribution of Constant Arguments

As an example of how to read Table 3.1, notice that row 10 lists the numbers of functions which have 0/10,1/10, 2/10, .., 9/10,10/10 constant arguments. We assume the number of parameters of all the functions does not exceed 10 for the analyzed code. In our implementation, we choose to test C programs of different sizes and styles. Some of them have hundreds of function calls in a single file.

We designed a large size of output matrix for the relatively large programs. Here we use an example of a 10 * 10 matrix Table. Element Table 3.1[i,j] reports the number of invocations that have i parameters with j of them constant. The sum of Table[i,i] (for $0< i <11$) is the number of functions which have all parameters constant. For example, the value of Table 3.1 entry [5,2] is 10, which means there are 10 function calls that have 5 parameters with 2 of these parameters constant.

The significance of this information is that the data reflects the frequency of distribution of constant parameters. This is also a significant measure of the benefit that we can gain from applying partial evaluation.

3.3.2.3 How many specialized versions would be obtained ?

One of the basic strategies of partial evaluation is to produce specialized functions by residualizing calls. The partial evaluator propagates constant values and folds constant expressions, produces specialized versions of the

values and folds constant expressions, produces specialized versions of the source function. It is possible to generate more than one specialized version for each source function. For example, for the function work(int x, int y, int z), if there were four function invocations:

(1) work (2, a, b); (2) work (2, 3, c);

(3) work (d, 3, e); (4) work (g, 3, h);

There would be three specialized versions obtained from the above function calls:

(1) work_2(int x, int y);

(2) work_2_3(int x);

(3) work_3(int x, int y);

There are two types of partial evaluator. A *monovariant* partial evaluator produces at most one specialized function for every source function. A *polyvariant* partial evaluator can produce many specialized versions of a source function. Our project collects the maximum number of specialized versions of source functions for the latter and for more complicated cases. So another important data item that we want to collect is the number of specialized versions that could be obtained.

To determine if a function call should be specialized, we need data on:

(1) How often the same constant parameter has the same value; and

(2) How often it is the only constant value.

Therefore, the statistical data we need to collect from the C code is:

(1) The total number of function calls;

(2) The total number of functions;

(3) For each function, the number of function invocations with the same number and value of constant parameters.

We can determine the number of specialized versions according to the data we collect.

3.3.2.4 The Number of Unevaluated Expressions and Functions with Unevaluated Expressions

As we state in the previous section, rather than computing the value of an expression parameter, we output the whole expression as a parameter to a function call and judge if it is constant. This might cause errors when we collect data on function calls with the same constant parameters. Let's consider the following two function invocations.

(1) work(4,x)

and (2) work(2+2,y)

Our program will treat them as two invocations of function *work(a,b)*, with different constant values for the first parameter, 4 and (2+2), respectively, though these two in fact have the same constant value for the first parameter after fully evaluating the expression of the parameter. According to the report produced by our program, we might have a different

number of function specialized versions and therefore jump to a different conclusion of the benefit of applying partial evaluation.

That is the reason why these data need to be collected. Due to the complexity of interpreting C code for typical systems applications, we do not evaluate the expressions at this point. In practical C code there is also little chance that function calls will have different constant forms of the same argument. We collect the number of these unevaluated expressions in order to see how much it affects the accuracy of our conclusions. If the fraction of unevaluated constant arguments is small then our conclusions are valid. If the number is large then a more complex statistics gathering program would be needed. As is shown in Table 5.1, the number of unevaluated constant arguments is a small fraction of the total number of constant arguments, and hence the effort required for full evaluation of all constant arguments is not warranted.

Chapter 4

Data Collection Method

4.1 Introduction

In chapter 3, we state what kind of statistical data were collected. Our methodology to do that data collection is to modify an existing C recognizer to read a C program, interpret the code, and report to a file the information about each function call invoked in the input program. With this file of function calls, we analyze the functions and their parameters, and get the statistical data we need.

The following sections will illustrate the design and the implementation details of the project.

4.2 The Analyzer

A program called "Analyzer" was developed to collect the statistical data. The purpose of this program is to read the C code, and report every function

invocation and its parameters. To do this we modified an existing C recognizer coded by members of the Programming Language Design and Implementation Group at the University of Manitoba.

The C recognizer, Crec, was developed to a part of a source-to-source translator for the Safer_C project. Its main task is to read a program, build a parse tree for it, determine if it is a C program, and give error messages if there are any syntax errors.

Crec uses Lex and Yacc to generate a scanner and a parser. It reads the input program, breaks the input into tokens using the scanner, and parses the token stream according to the syntax rules of the C language. In the parsing phase, it also produces a parse tree which we can think of as an image of the program. In other words the parse tree reflects the structure of the program. Our program, Analyzer, will process the C code according to the parse tree, retrieve all the information on every function call, report them to a text file, analyze the information in the file, and get the statistical data. According to these data it estimates how much effort would be needed to apply partial evaluation to the input program and how much benefit would be gained. By running the program "Analyzer" against the representative C files that we selected, we come to a conclusion on the benefit of applying partial evaluation to C code.

Since the program "Analyzer" uses Lex & Yacc and standard compiler construction techniques to generate the parse tree, the following section is a

brief review of the two widely used compiler generation tools, Lex and Yacc.

4.3. Compiler Construction

The following diagram illustrates the five phases of compilation.

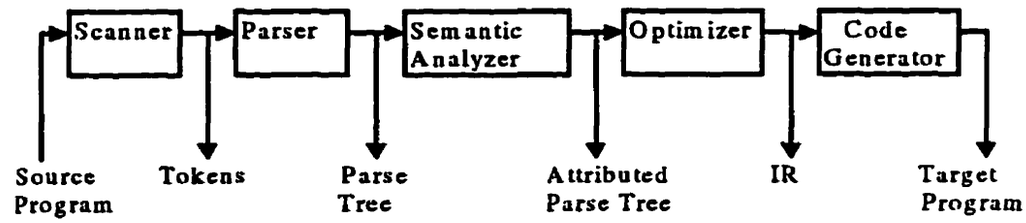

Fig. 4.1 Compiler Construction

To aid in the generation of compilers, a lot of compiler tools were developed. Lex and Yacc are two well known tools to generate scanners and parsers, respectively.

As a scanner generator, Lex accepts a high-level, problem oriented specification for character string matching, and produces a function (the scanner) that recognizes tokens described by regular expressions. The regular expressions are specified by the user in the source specifications

given to Lex. The scanner generated by Lex recognizes strings described by these expressions in an input stream and partitions the input stream into strings matching the expressions.

Yacc is a parser generator. The Yacc user prepares a specification of the parsing process; this includes rules describing the input structure, code to be invoked when these rules are applied, and a low-level routine to do the basic input. Yacc then generates a parser, which calls the user-supplied low-level input routine (the scanner) to pick up the basic elements from the input stream. These tokens are organized according to the input structure rules, called grammar rules; when one of these rules has been recognized, the user code supplied for this rule, an action, is invoked. Actions have the ability to return values and make use of the values of other actions.

In our project, the input user-specification for Lex is regular expressions describing C tokens, and the input for Yacc is the grammar rules for ANSI C and the code invoked for each rule, which generates the parse tree.

4.4 Parse Tree

As we have mentioned, the outcome of the parser is a parse tree. The way the parse tree is generated is that we put the code for parser tree generation in the action part of the Yacc grammar. When the right part of one of the grammar rules is recognized, the code is invoked to generate a

node in the parse tree.

For example, as defined in the file "min_parse_tree.h" in the package "Analyzer", each node in the parse tree consists of three parts:

(1) Code : The code for grammar symbol represented by this node.

For example, "ac_func_call" is the code representing a node in the parse-tree which stores the information of a function invocation.

(2) Value: For a nonterminal node, the value is a pointer to the first descendant of this node. Remaining descendants are in the sibling list of the first descendant. For example, if "ac_func_call" is a nonterminal node, then its value is its first descendant node "ac_func_name." Remaining descendants like "parameter_list" which represents the parameters of the function are in the sibling list of this value node.

For a terminal node, the value field may contain a text string giving the actual characters in the terminal symbol.

(3) sib: A pointer that points to the next sibling of this node in the parse tree.

Therefore, for a function invocation work(2, b), the corresponding part in the parse tree will look like:

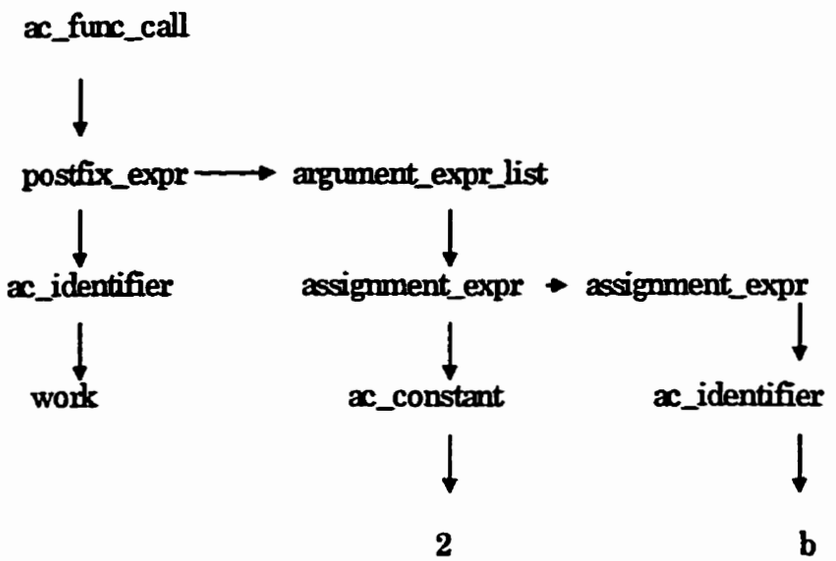

Fig. 4.2 A Subtree in the Parse Tree

As shown above, the parse tree contains all the information of the program. By traversing the parse tree, we can get information about every function call, which includes the values of the actual arguments of the function, the loop nesting depth of the function call, etc. We will explain important design decisions we made during the implementation of the project.

4.5 The Design of Analyzer

Following is a schematic diagram of the program Analyzer.

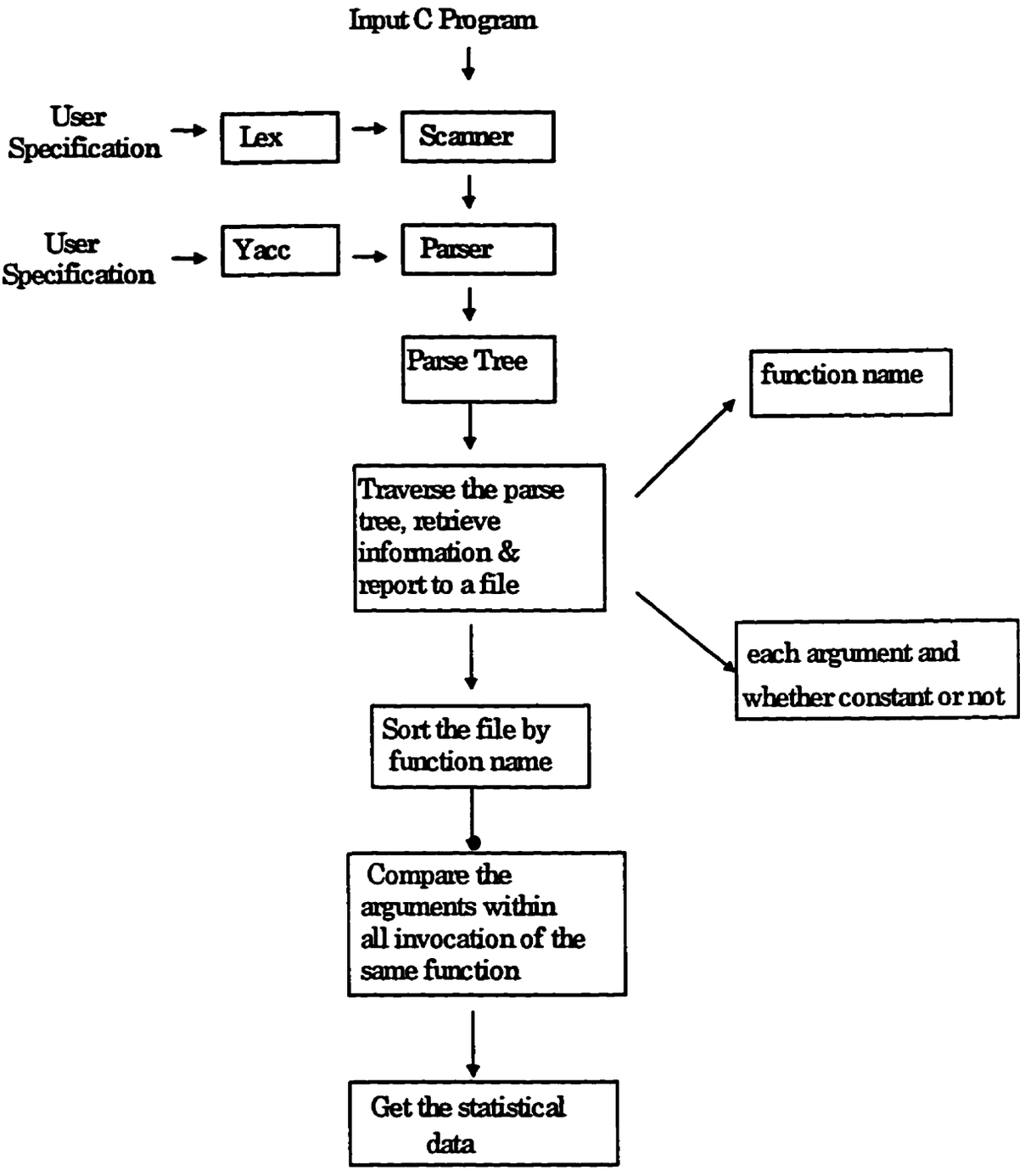

Fig. 4.3 A Schematic Diagram of Analyzer

4.5.1 Traversing the Parse Tree

Once the input program has been parsed, a parse tree which stores the whole program in its own structure is produced. By traversing the parse tree, all the information of the program, including every declaration, every statement, etc, can be examined. For this project, we need all the data on function invocations; therefore in the program "Analyzer," the whole parse tree is traversed, and the information is retrieved whenever a node related to a function call is encountered. Basically, for a C program the parse tree consists of two parts. The following section will describe how the two basic kinds of nodes are being analyzed to get the information we need.

(1) The declaration part.

The declaration part is the code which includes all the declaration before the main() function in a C program. Consider the following example.

```
int  a = sin(0.5);
```

This declaration includes a function invocation sin(0.5), and data on this function call needs to be collected too. Therefore when traversing a parse tree, and a node which represents a declaration is encountered, the program "Analyzer" examines the subtrees of this node and checks if there are any

function invocations.

(2) The function definition part.

The other part of the parse tree that needs to be checked is the nodes representing function definitions. Function invocation could happen in expressions, variable declarations in the definition of the functions, statements, etc.

4.5.2 Function Invocation Nodes

When traversing the parse tree, every time a node that represents a function invocation is encountered, all the information about the function call has to be retrieved, which includes the function name, the actual arguments, etc. In this project, a function invocation node has the following format:

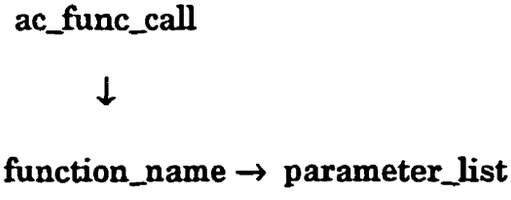

Fig. 4.4 A Node of Function Invocation

In this structure, the name of the function, which is stored as a string in this project, is the first descendant of the node of "ac_func_call." Its sibling is

the argument list of this function call.

4.5.3 The Function Name and the Argument List

To extract the function name from the parse tree, the value of the first descendant of the node "ac_func_call" is retrieved, and is output to a text file which has been opened for this purpose. The next step is getting the arguments.

An argument of a function call in C code is an expression that could be a variable, a constant, an assignment, a function call, etc. For example, we might have a function call such as:

work_a(2*3 +5, 4, x+1);

For those arguments that are constant expressions, we decided to output the whole expression instead of partially evaluating them. For the above example, the arguments being output will be:

(1) 2*3+5; (Not 11) Constant Value;

(2) 4; Constant Value;

(3) x+1; Non-Constant;

The reason that we choose to do so is that it is too difficult to interpret all possible C expressions correctly with proper type promotions, word size, etc. As shown in Table 5.1, we later found that unevaluated constant expressions would only have a small effect on the results collected. Though we haven't

evaluated the value of an expression, we still judge if it is a compile-time constant value. The method to judge a constant argument will be explained in section 4.5.7.

Another case of tricky arguments is when the argument itself is a function call.

Consider the following function call:

```
work_b(x, do_it(sin(y)), cos(0.5));
```

For this function call, some of its arguments are nested function calls. We need to recognize and output it correctly, and determine if it is a constant value. The arguments to work_b() that we get will be:

(1) x; Non-Constant Value;

(2) do_it(sin(y)); Non_Constant Value;

(3) cos(0.5); Constant Value;

This example is treated in more detail later.

4.5.4 An Output File prg.stat

To gather the information on function calls collected from a parse tree, a text file is created. This file is opened prior to the data collection. Its name will be of the form *prg.stat* which inherits the name of the input C program *prg.c* but with a different extension .stat. The *stat* is short for "statistics." Each line in the file *prg.stat* contains the information for one function call, which includes

(1) The name of the function,

(2) each argument of the function, and

(3) whether or not that argument is a constant expression.

When the data collection has been finished, the file will be sorted by function name and the data in it will be analyzed.

The following figure shows the output lines for function call work_a & work_b shown previously, in the file *prog.stat*.

```
@ work_a @ 2*3+5 $ 4 $ x+1 #
@ work_b @ x # do_it (sin(y)) # cos (0.5) $
@ do_it @ sin (y) #
@ sin @ y #
@ cos @ 0.5 $
```

Fig. 4.5 Sample Lines of the File *prog.stat*

4.5.5 A Recursive Algorithm for Retrieving Data on Function Calls

The name of the function is stored as the first descendant of the node of the function call. Its value which is stored as a string is extracted and is output to the file *prg.stat*. To collect the data on the arguments, a recursive algorithm is used.

In a C program, the arguments of a function.call are expressions that can be a variable, a numeric, a function call, an assignment statement, etc. In the input specification to the parser generator YACC, i.e., the program *parser.y*, expressions are defined recursively:

```
expr
        : assignment_expr
        | expr ',' assignment_expr
```

And the *assignment_expr* could be one of several kinds of expressions:

```
assignment_expr
        : logical_or_expr
        | logical_or_expr '?' logical_or_expr ':' conditional_expr
```

Therefore, when we decode a parse tree, trying to extract the expression of the argument to the output file *prg.stat*, a recursive algorithm and a *case* statement are used.

The *case* statement lists all the possible forms that an expression could take, i.e., constant, identifier, assignment, function call, etc. For some simple cases, like identifiers, just simply getting the string value of this node and outputting it to the file *prg.stat* is required; for some complex cases, such as when the expression is an assignment expression, this assignment expression has to be retrieved recursively to get the whole expression. Actually, recursive calls have to be used for most of the branches in the *case* statement.

4.5.6 A Linked List of the Function Calls

The recursive algorithm is not enough for our task. For most of the cases, the argument expressions can be extracted correctly to the output text file with this algorithm. But as we mentioned before, consider the function call

work_b(x, do_it(sin(y)), cos(0.5));

In this function call, the second argument *do_it(sin(y))* is a nested function call. In our point of view, it is not only an argument of one function call, but also another function invocation itself; that is *sin(y)*. The original recursive algorithm can output this argument to the file correctly, but does nothing to deal with collecting the argument data for a function call which itself is also an argument to a function call.

To solve this problem, we use a simple but elegant solution: Record the pointer to this function_call_argument node in the parse tree, keep a linked_ list of these kinds of nodes, and handle them using the recursive algorithm when the whole program has been processed and the data for the first level of call has been collected. The following figure illustrates the linked _list structure.

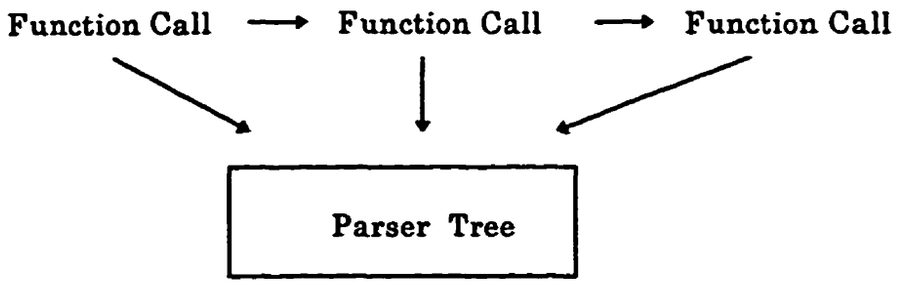

Fig. 4.6 Linked_list of Function Calls

Each node in the list consists of two pointers: one points to the next node in the list, another pointer points to the node of the function call in the parse tree. With this linked list, we will not miss any of the function calls in the program.

For example, for function call work_b(x, do_it(sin(y)), cos(0.5)), the analysis program outputs "work_b" as the function name; and "x" as its first parameter, "do_it(sin(y))" as its second parameter, and "cos(0.5)" as its third parameter. The analysis program also realizes that the second and the third parameters are function calls also. Therefore "do_it(sin(y))" and "cos(0.5)" will be sent to the linked_list, and will be processed using the recursive algorithm to extract the information on these two function calls.

4.5.7 Recognizing Constant Parameters

An important job to do for the data collected is to judge if an argument is a constant argument. Due to the varieties of expressions, a recursive

algorithm is used. For an argument that is an expression involving operators, rather than a simple factor, since we do not evaluate the expression, we will decide if it is a constant value according to the value of its components. If they are all constants, then the expression is considered to be a constant argument. For example, *3*8+12* is considered to be a constant value, while *3*a+12* is not.

For an argument that involves a function call, we assume there are no global variables involved. Therefore if all of the parameters are constants, it itself is a constant argument, otherwise it is not. For example *work(2,3,4)* is considered to be a constant argument but *work(2,3,t)* is not.

4.5.8 Analyzing the Data File

After traversing the whole parse tree, and reporting all the information to the stat file, we start to analyze the file. This includes the following steps.

(1) Sorting the file.

We use the Unix utility "sort" to sort the file by the file name. That is a new experience for me to call "sort" within a C program. In this project, we assume that functions are identified by their names. Function calls with the same name are considered to be the same function. A sorted file consists of function calls listed by their names.

A problem arises when only a pointer to a function is specified in the function call instead of a function name. We decide to treat all such function

calls as being to the same function because it is not possible to get the function name by a dynamic pointer.

(2) Collecting the Data Distribution.

When a sorted file is ready, the next step is to collect the distribution of the data; that is, how many functions have 0, 1/n, 2/n, 3/n, .., n-1/n, n/n constant parameters (n is the number of the parameters). The output of this distribution is a 2-dimensional matrix A[i,j]($0 \le i,j \le n$) which we presented in chapter 3. We also can get the number of functions with all arguments constant from this matrix. This number is the sum of A[i,i] for i =0 to n.

(3) Determining the number of the specialized versions of a function.

A function might be called a few times with different numbers and values of constant parameters. Each of these combinations would generate a different specialized version of a function. To determine the number of specialized versions of a function that would be produced, more analysis needs to be done to the file. This is a tricky part in terms of programming because the information (function name, parameters, constant value, ...) of each function invocation is stored in the text file as a string. The hard part is to extract each part of this information correctly from each line.

4.6 Implementation of the Project

The project is implemented on Sun/Unix, using Lex, Yacc and the C programming language.

Chapter 5

Results

5.1 Introduction

The purpose of this project is to collect statistics on the use of translation-time constant arguments in C language source code. Our analysis program takes a C source program as input, and produces the statistical analysis as the output. We analyzed the source code from three important projects coded in C: the X-Window Library, the LINUX operating system, and the Gnu C compiler. The following sections describe how we ran the analysis program, report the results we obtained, and present some discussion of the results.

5.2 Software Selected for Analysis

In the project, we analyze the source code of the following software.

(1) The X Window System. This code was initially developed at the

Massachusettes Institute of Technology. It is now widely used as a graphical window system on UNIX systems for a large number of varied platforms. The results are reported for version 11 release 5.

(2) The LINUX operating system. This code was initially developed by Linus Torvald in Norway. It is now maintained and enhanced by a large number of programmers scattered all over the net. It is widely used to provide the UNIX environment on personal computers. The results are reported for version 1.1.59.

(3) The Gnu C compiler gcc. This code was developed by the Free Software Foundation and has been maintained over the years by a variety of programmers. It provides a reliable C compiler for a large number of target architectures and operating systems. Results are reported for version 2.7.2.

The reasons for choosing the source code listed above are:

(1) The source code was developed by different implementers. We don't want to select source from a specific group of implementors which might have a specific style and may not be a representative of the C code in general.

(2) The source represents typical applications of the C language, namely: graphical user interfaces, operating systems, and compilers.

(3) The source code is well known and widely used.

(4) The source code is available over the Internet.

5.3 Data Sampling

The source code analyzed had a high degree of complexity that is typical of systems programming. It can take years of work to perfect a language processor that correctly handles code with such a high-level of complexity. In fact, the software systems that were selected for analysis often specify the exact version of the compiler that must be used in order for the software to run correctly. They also often use special features of the gcc compiler that are not part of ANSI C.

As a result, we had two options:

(1) Engage a long-term coding effort to perfect a statistical analysis program that could accept all this software correctly as is, or

(2) Hand modify the input code so that it could be acceptable to a simple program.

Since the objective of our project is a single-usage data analysis program rather than a general purpose software package, we chose the second alternative. The changes to the input programs were carefully made to ensure that the statistics being collected would not be affected. This requirement for hand manipulation limits how much source code we could reasonably analyze. Our solution was to sample the input data by selecting one of every n source files for analysis. The sampling rate for each input project is shown with the statistics.

Even with this reduced code sample size, many weeks of careful editing were required to make the selected software acceptable to our translator.

5.4 The Data Collection Method

The data collection process consists of the following steps.

(1) Preprocessing

Before a source file can be analyzed it must be run through the C preprocessor "cpp". The reason for this is that a C program cannot be parsed correctly until the transformations described by the preprocessor directives are applied. In this step, the symbolic constants represented by preprocessor symbols are replaced by literal constants.

In many cases, this step was not straightforward because the source code analyzed is highly portable low-level code. Sometimes the generated C source lines were too long to be loaded by a text editor such as the UNIX editor "vi", or to fit in the limited buffer normally provided for "lex" generated scanners. The ability to inspect and make small changes to the source code was vital to allow the continued processing of source code that was otherwise rejected by our parser. A simple "preprocessor postprocessor" was coded to edit the output of preprocessor and break it up into manageable lines.

(2) Run the Analyzer

To get the statistical data on the C source, the analyzer program is run

on the C source program. The source program is parsed into a syntax tree. The syntax tree is traversed recursively locating function invocations. Information about each function call is collected and output to a file, and the file is then sorted by the function name. At the same time, our analyzer program also collects statistical data on functions in the source program and reports them to standard output. Hence for each source program, the analyzed results consists of:

(1) A text file which contains all the data we need on function calls: the name of the function and all the parameters of each function call, and whether or not they are constant expressions.

(2) Statistical data reported on the screen based on the text file. The data includes: a matrix which indicates the distribution of constant parameters of function calls in the input program, the maximum number of specialized version of function calls that would be needed, the total number of functions, the total number of function calls in the program, the number of unevaluated argument expressions and the number of functions with unevaluated argument expressions.

5.5 Results

The following tables report the results for each package analyzed.

		X-Windows	LINUX	gcc	Total
1	Source file sampling rate	1/5	1/6	1/8	
2	No. of source lines analyzed	11,396	16,018	33,024	60,438
3	No. of function calls	616	3,227	3,803	7,746
4	No. of functions invoked	218	527	690	1,435
5	No. of calls with all constant arguments	4 0.65%	141 4.4%	120 3.1%	265 3.4%
6	No. of specialized versions	35 16%	504 96%	269 39%	808 56.3%
7	No. of unevaluated constant Expressions	2 0.3 %	65 2 %	4 0.1 %	71 0.1 %
8	No. of function calls with unevaluated constant	2 5.1 %	62 12 %	4 0.6 %	68 4.7 %

Table 5.1: Statistics on Function Calls

5.5.1 Discussion of Table 5.1

Row 5 in Table 5.1 shows that for at least two of the packages, LINUX and gcc, supporting the full evaluation of functions with all constant arguments could be an important feature of a code improver. It would reduce up to 4.4% of function calls to their minimum possible implementation. At the very least, that row indicates that it is warranted to do a further analysis of which functions have side effects.

Row 6 shows how many extra function versions would be produced if functions were specialized for every combination of unique constant

arguments in all invocations. This result is interesting because it shows that the total number of functions would not increase by more than 96%, and more commonly by less than that. This could be an acceptable price to pay for higher execution speeds.

Rows 7 and 8 show how much error was introduced by the analysis program due to the fact that it did not compare the binary values of constant arguments, but rather compared their lexeme strings. These rows show that the error is at most 12%. Furthermore this error is toward the favorable side, since it means that possibly up to 12% fewer specialized functions would be generated. A cursory inspection however showed that in all likelihood there were no constant arguments with equal binary values but different lexemes.

This table also shows that there can be substantial differences in programming styles between programming groups. Thus code improvement techniques that benefit one project substantially may have little effort on other projects.

5.5.2 Ratio of Constant Arguments to Total Arguments

The following tables report the number of constant arguments versus the lengths of the argument lists. This information indicates the degree of specialization of a specialized version of a function. The more arguments that are constants, the more efficient will be the specialized version. An interesting fact represented in the table is that no function in the analyzed

projects had more than three constant arguments, regardless of the length of the argument list.

		No. of Constant Args				Total
		0	1	2	3	
No. of Args	0	6				6
	1	256	4			260
	2	119	7			126
	3	114	5	2		121
	4	27	8	1		36
	5	17	2	1		20
	6	8	14			22
	7	9			1	10
	8	1				1
	9	2				2
	10					
	11					
	12	4	2	4	2	12
Total		563	42	8	3	616

Table 5.2: X Window System

		No. of Constant Args				Total
		0	1	2	3	
No. of Args	0	41				41
	1	1312	112			1424
	2	687	542	27		1256
	3	218	100	28	2	348
	4	41	16	8		65
	5	45	8			53
	6	10	1	1		12
	7	3	1	2	1	7
	8	9	1			10
	9	4	5			9
	10	1				1
	11					
	12					
	13					
	14	1				1
Total		2372	786	66	3	3227

Table 5.3: The LINUX Operating System

		No. of Constant Args				Total
		0	1	2	3	
No. of Args	0	334				334
	1	1049	110			1159
	2	1252	123			1375
	3	614	78	7	20	719
	4	164	38	6	1	209
	5	34	44	6	5	89
	6	1	8	2		11
	7	7				7
Total		3455	401	21	26	3903

Table 5.4: The Gnu C Compiler "gcc"

		No. of Constant Args				Total
		0	1	2	3	
No. of Args	0	381				381
	1	2617	226			2843
	2	2058	672	27		2757
	3	946	183	37	22	1188
	4	232	62	15	1	310
	5	96	54	7	5	162
	6	19	23	3		45
	7	19	1	2	2	24
	8	10	1			11
	9	6	5			11
	10	1				1
	11					
	12	4	2	4	2	12
	13					
	14	1				1
Total		6390	1229	95	32	7746

Table 5.5: Totals for Three Input Projects Taken Together

Chapter 6

Conclusions

In this thesis, the principle of partial evaluation and its major applications are surveyed. Two important partial evaluation techniques are also discussed. Based on one of the basic strategies of partial evaluation – unfolding function calls, an analysis project was designed to collect statistics on the use of translation-time constants as arguments in C language source code. The project was implemented and source code from three well known projects coded in C was analyzed .

It was found that polyvariant specialization of all calls with some constant parameters is feasible on real system code. It can be an important feature of a code improver. It was also found that the effort of specialization can be made at an acceptable price for higher execution speed.

Bibliography

(1) Salomon, Daniel J., "Using Partial Evaluation in Support of Portability, Reusability, and Maintainability." International Conference on Compiler Construction CC'96. 1996.

(2) Charles Consel, Oliver Danvy, "Tutorial Notes on Partial Evaluation". Conference Record of the 20th Annual ACM SIGPLAN-SIGACT Symposium on principles of Programming Languages POPL'93, Charleston , SC, 1993.

(3) Uwe Meyer, "Techniques for Partial Evaluation of Imperative Languages". In *Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, Pages 145-164. ACM, 1991.

(4) Daniel Weise, Roger Crew. "Programmable Syntax Macros". 1993, ACM SIGPLAN Notices, V28, #6, pp156-165.

(5) Vivek Nirkhe, William Pugh. " A Partial Evaluator for the Maruti Hard Real-Time System", Proceedings of the Twelfth Real-Time System Symposium, San Atonio, Texas Dec., 1991. IEEE Comput. Soc. Press.

(6) Salomon, Daniel J., "C_Breeze: Syntactically Improving the C Language

(6) Salomon, Daniel J., "C_Breeze: Syntactically Improving the C Language for Error Resistance", Technical Report, University of Manitoba, 1995.

(7) Romana Baier, Robert Glück, Robert Zöchling, "Partial Evaluation of Numerical Programs in FORTRAN", PEMP '94, ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation, 1994.

(8) Andrei P. Ershow, "On Mixed Computation: Informal Account of the Strict and Polyvariant Computational Schemes", NATO ASI Series, Vol. F14 Control Flow and Data Flow: Concepts of Distributed Programming. Edited by M. Broy, Springer-Verlag Berlin Heidelberg 1985.

(9) Lars Ole Andersen, "Partial Evaluation of C and Automatic Compiler Generation".In *Proc. of Compiler Constructions – 4th International Conference*, CC'92, pp. 251 – 257, Springer-Verlag, October, 1992.

(10)Donald Michie, "'Memo' Functions and Machine Learning", NATURE,Vol. 218, April 6, 1968.

(11)Vivek Nirkhe, William Pugh, "Partial Evaluation of High-Level Imperative Programming Languages, with applications in Hard Real-Time Systems", Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL'92, 1992.

(12)Neil D.Jones, Peter Sestoft, Harald Sündergaard, "An Experiment in Partial Evaluation: The Generation of a Compiler Generator", First

International Conference on Rewriting Techniques and Applications, May 1985.

(13) Neil D. Jones, "An Introduction to Partial Evaluation", ACM Computing Surveys, Vol. 28, No.3, September 1996.

(14)Charles Consel, Francois Noel, "A General Approach for Run-Time Specialization and its Application to C", POPL'96, St. Petersburg FLA USA, 1996.