A MULTI-TECHNIQUE C INLINER

by

Chengyan Zhao

B.Sc. Computer Science, Peking University, 1995

A Thesis Submitted In Partial Fulfillment of
the Requirements for the Degree of

Master of Computer Science
in the Faculty of Computer Science

Supervisor:	O. Kaser, Ph.D. Computer Science
Examining Board:	J. DeDourek, MS. Computer Science, Chair
	W. Du, Ph.D. Computer Science
	B. R. Petersen, Ph.D. Electrical and Computer Engineering

This thesis is accepted

Dean of Graduate Studies

THE UNIVERSITY OF NEW BRUNSWICK

June, 1998

Abstract

Function inlining has been considered an important optimization method for compiled languages. In this thesis, we especially study the inlining benefits and present an object-oriented implementation of a source-level C language inliner. Various parsing and inlining topics, such as: message driven system, design patterns, parse-tree and symbol-table organizations, memory management strategy, program standardization and function-inlining steps are carefully presented.

For efficiency and ease of implementation, many optimizing compilers implicitly impose an "inlining-decision algorithm" to restrict the conditions under which a function invocation could be inlined. Source-level inlining, profile-guided inlining, cache issues and the version issue have lead to various conflicting decision algorithms. To overcome previous inlining work, an ambitious inlining-decision algorithm combined these techniques, which creates an automatic C inliner.

CONTENTS

LIST OF FIGURES

ACKNOWLEDGMENTS

I would like to express my very deep and sincere gratitude to my supervisor, Dr. Owen Kaser, for his patience, caring, attention and unlimited assistance that is always ready and guides me through the entire thesis preparation and master level research.

I would like to thank my family, my Mom, Dad and younger sister, for their caring, support and encouragement, which gives me enough confidence to face the challenges and sustains me through the academic years.

I also would like to thank Dr. Jutta Degenar (Computer Science, Berlin Technical University), Mr. Ed Mulroy (Inprise Corporation, formerly Borland International), Mr. Doug Robbins (Technical support team, Microsoft Corporation), Mr. Mark M. Lacey (Visual C++ backend team, Microsoft Corporation), Mr. Jeffrey Reilly (Intel Corporation, Compiler Group), Mr. Robert D. West (Hewlett Packard, Compiler Group) and all those people whose names I cannot clearly remember, who have friendly helped me and patiently answered my questions.

Chapter 1

Introduction

In recent years, various studies [5,7,10,13] have been undertaken to determine the efficiency of a compiler's optimizations. As part of a compiler's back end, optimizations are always considered a necessary and important part of the language implementation. The optimization techniques generally include inlining, common-subexpression elimination, dead-code elimination, loop motion, tail-recursion removal, and so forth [21,22]. Results of various studies have shown that current optimization techniques are indeed worthwhile for general-purpose programming languages [13]. Most compilers' individual optimizations have a fairly small effect, and generally range from 4% to 12% [7,10,22,26]. This makes their cumulative effect important. Previous research [6,10,12] showed that the inlining optimization could improve program execution speed up to 6%, a significant contribution. As one of the important optimization techniques, inlining is generally applied first in the sequence of compiler optimizations [7,10]. This implies that a source-to-source inliner that applies the optimization even before compilation is thus similar. In this thesis, we describe the details of how to effectively inline C programs and demonstrate the benefits that could be gained from *better* inlining.

1.1 Overview of Inlining

The function-inlining optimization (in short, inlining) involves the replacement of a callsite by the code of the called function, with suitable substitutions for parameters and conflicting identifiers [6]. Expanding functions inline has a number of advantages. The technique has been used for improving the execution speed of software [5,6,21], since it eliminates unnecessary and expensive function-call-and-return overhead. Surprisingly, this technique may also improve the space efficiency, depending on the number of times the function is invoked, the size of the function and the size of possible invocation overhead saved. Further, inlining allows and encourages programmers to use functions for software-engineering purposes, such as clarity, modularity and maintainability, without making the program suffer from execution slowdown.

The goal of our thesis is to build an automatic C inliner. We carefully study the details of inlining and its effect on program size and execution efficiency. Moreover, we implement a source-to-source inliner that reads in a C source program and produces the inlined C source code for compilation and execution. To serve as a stand-alone inlining tool, the inliner strictly follows the ANSI C [4] standards with some GCC [24] extensions[1]. The system discussed in this thesis is pattern-directed, message-driven and object-oriented. It consists of around 20,000 lines of C++ source code and currently supports both Win95 and Linux [25] platforms. Portability is achieved by use of the ANSI C++ 2.1 standard only, without any implicit use of machine-specific information or compiler-dependent class libraries.

[1] This includes nested struct/union declarations, macros left after C preprocessing *(#line* directives, *#pragma*, etc) and gcc extended keywords such as __inline__, __const__, __absolute__, and so forth.

1.2 Inlining at Different Levels

Inlining can be done in several different ways. One is to inline a source program and produce a modified source program. This is superficially similar to textual replacement. An alternative is to apply inlining on the intermediate code that many compilers use when optimizing. It can even be done on the executable by the loader of the operating system. We next analyze each technique and give an explanation for our choice.

1.2.1 Source-Level Inlining vs. Textual Replacement

From the description of inlining, one might think that source-level inlining is as simple as textual replacement, where functions' definitions are recorded from source code and the textual replacement happens at the callsites accordingly. An example where this over-simplified idea works is given in Fig. 1.1.

It is acceptable to do textual replacement in this particular example, because the function callsite is in a standalone statement (not nested in another construct), plus the function's definition has neither parameters nor return value. However, things are not always this easy.

Consider another example, where a function callsite happens to appear inside an *if_then_else* conditional construct, as shown in Fig. 1.2. This time, the invoked function has parameters and a return value. In this situation, it is impossible to do inlining by just making textual replacement at the callsite, because ANSI C [4] syntax does not allow curly brackets and commands in expressions. Curly brackets can only appear in compound statements and array initializations. More sophisticated compiler work is required.

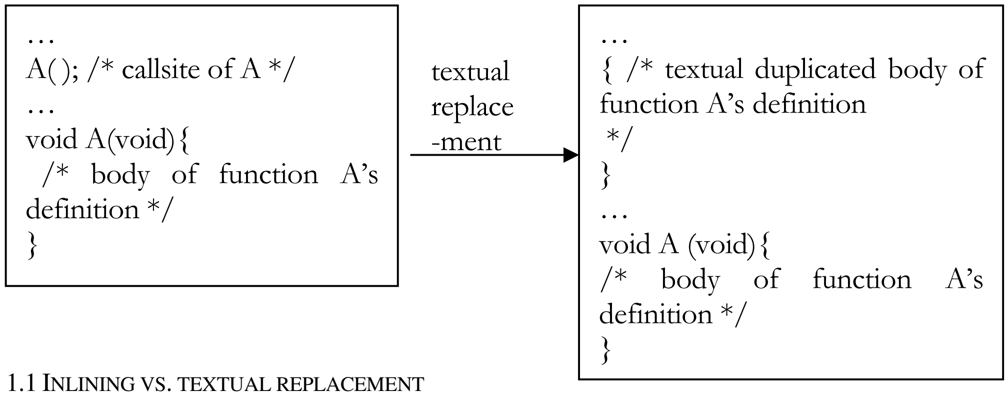

FIG. 1.1 INLINING VS. TEXTUAL REPLACEMENT

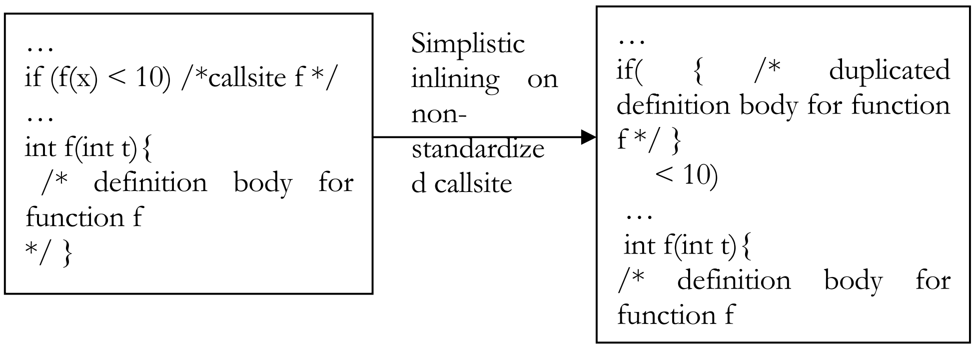

FIG. 1.2 FAILURE OF TEXTUAL REPLACEMENT

1.2.2 Object-Level Inlining

Object-level inlining replaces a callsite with the corresponding function's intermediate code or object code and thus all possible adjustments for conflicts also must be made at the intermediate level. Gcc [24] uses this approach for its limited inlining and so did an earlier version of EQUALS [11,23]. However, this inlining technique requires that intermediate code must be available. It is also impossible to make a general intermediate-level inlining tool that everybody could use, because the intermediate-code specification is not standardized. For this reason, object-level inlining is out of the scope of the thesis.

1.3 Issues in Inlining

There are several issues directly related with inlining, such as callsite structure, inlining steps and inlining decisions. We next give each an introduction.

1.3.1 Structural Standardization

Due to the nature of the C programming language, a callsite can appear almost anywhere in the source code. It could be a standalone callsite, not nested in any statement. It could be nested in an expression or other function callsites. Moreover, it could be part of a complex expression controlling a conditional evaluation or part of a loop control expression or a part of an expression with short-cut operators. Effort must be made to convert all callsites into a standalone format, which is the only possible form suited for inlining.

1.3.2 Inlining Adjustments

A source-level inlining technique replaces the selected function callsites with its corresponding function definition. Necessary adjustments must be made to parameters and identifiers of the duplicated function body. This usually includes simulating parameter passing, detecting and renaming parameter-name conflicts, simulating *return* statements, relabeling function and recording return values. The adjustments guarantee that the inlined program is both syntactically valid and semantically equivalent to the code before inlining.

1.3.3 Inlining-Decision Algorithm

The inlining-decision algorithm is used to decide which callsite should be inlined and when the inlining should stop. It is important to realize that not every callsite is inlineable. Moreover, not every inlineable callsite should actually be inlined. Since inlining replaces the callsite with the called function's body, it makes a trade-off between the program size and the actual number of instructions executed. Before inlining is done to the standardized callsites, decisions must be made about which callsites should actually be inlined and which should not. After consideration of several previous researchers' work, a greedy inlining-decision algorithm is introduced. It combines several previously proposed techniques that have not been used together before. This inliner is ambitiously designed to overcome all previous inliners' achievements.

1.4 Structure of the Thesis

The entire thesis is structured as following.

In Chapter 2, some background information on related inlining work, such as descriptions of source-level inlining steps, is introduced.

In Chapter 3, we discuss the design and implementation details of an object-oriented parser. This includes the design of parse-node classes, development of a message-driven system, use of design patterns and description of miscellaneous parsing techniques.

In Chapter 4, we discuss the implementation of inlining techniques in detail. This includes the callsite-standardization process and the actual step-by-step inlining work applied for each selected individual callsite. Some of this expands on techniques sketched first in Chapter 2.

In Chapter 5, we present the implementation of a greedy inlining-decision algorithm that considers versions, profile information, code expansion, cache behavior and specialization opportunities.

In Chapter 6, we conclude the thesis work and discuss some possible future enhancements.

Chapter 2

Background and Related Work

Inlining is the technique to substitute a function callsite with its corresponding function's body, with necessary adjustments to make it syntactically valid and semantically correct. This chapter introduces related work and previous researchers' achievements and considers how one inlining system might combine these different achievements.

2.1 Parsing

Lexical analysis and parsing involves reading the source code, rejecting programs that are syntactically invalid, recognizing syntactically correct statements and constructing a parse tree. Sometimes, one distinguishes between *concrete* syntax and *abstract* syntax. The syntax specified by a conventional context-free grammar is called a concrete syntax, because it describes the exact syntactic structure of a program and its phrase structures. Abstract syntax is only concerned with the hierarchical relationships of forms and phrases, which are used to categorize the syntactic structures that exist. It need not worry about the exact details of program representation or how substructures interact. We will give details in Chapter 3 to see how to build a parser using concrete syntax to represent the ANSI C grammar.

2.2 Inlining at the Source Level

Davidson and Holler [5,6] described the details of source-level inlining for the C programming language. They implemented a source-to-source filter that accepted a C source program as input and produced inlined C source code after processing. An easy example of inlining is given in Fig. 2.1 and originally appeared in [5]. It demonstrates most of the inlining technique details they presented.

As shown, there are several necessary issues that need to be discussed. First, the function's called body is duplicated and the original callsite is removed. Note that the code expanded in place of a specific callsite is a duplicate of the callee's body, which can be obtained from the source program. It serves as the starting point for all the following techniques and adjustments.

```
…
t = strlen(s); /* callsite */
…
int strlen(char * s){
/* function strlen definition
*/
  char * p;
  int i = 0;
  p = s;
  while (p++)
    i++;
  return i;
}
```

inlining →

```
…
{ { int AA00001; /* return value */
    char * temp_s = s;
     /* parameter simulation */
     char * p;
     int i = 0;
     p = temp_s;
    while (p++)
       i++;
    { AA00001 = i;
     /* "return" statement simulation
         */
     goto exit_01;
     }
  }
  exit_01: t = AA00001;
  }
…
int strlen (char * s){
/* definition of strlen */  }
```

FIG. 2.1 SOURCE-LEVEL INLINING

Second, the optional *return* variable is created. If the function happens to have a non-void return type, the return value must be recorded no matter whether the original callsite's return value is recorded in the source code, because the return value needs to be recorded when replacing the return statement. If the function's return value is recorded in source code, then return it. Otherwise, it is simply ignored. A variable (shown in the figure above, "AA00001") is created with the same return type as the function and a distinct name controlled by a global counter is created and serves to record the function's return value.

Third, parameter passing is simulated. In order to execute a function call, the compiler needs to generate code to construct an activation record and pass all parameters into the record before calling the function. In the inlined case, there is actually no need to construct the activation record and pass parameters, but the parameter passing needs to be properly simulated by local-variable assignment. Each actual argument appearing at the callsite must be assigned to a local variable inside of the cloned function body. This simulation needs to take place earlier than the occurrence of the function's local variables to maintain the original order of evaluation and hence guarantee that data dependencies are properly maintained.

Fourth, identifier-name conflicts are detected and renaming is applied. There is a possibility that an actual argument has the same name as one of the local-variable names used in the cloned function body. A name-conflict detection and renaming method is necessary. Unlike Davidson and Holler [5], we do not use a method to explicitly check for such name conflicts. A simpler idea is used to deal with it instead, and details will be given in Section 4.7.

Fifth, return statements are replaced. Clearly, it is impossible to have any return statement inside the cloned function body after inlining, because we do not want the callee's cloned return statements to cause an immediate exit from the caller. All return statements in the cloned function body must be removed and replaced with an assignment statement, followed by a "goto" statement. These statements are used to write the actual returned value into the return variable and jump to the exit point of the cloned function body, which properly simulates the behavior of the function's return operation.

Sixth, an exit label is created. Theoretically speaking, every function must return, no matter whether the function actually returns a value or not. This implies that every cloned function's body must have an exit point to properly simulate the function's return and this exit point always appears at the very end of the cloned function's body. This creation is done by automatically generating a distinct label and inserting it at the very end of the cloned function body.

Seventh, the return value is optionally recorded by the original caller. If it is recorded, an assignment statement must be inserted after the exit label, and it puts the return value calculated by the callee into the variable that actually holds the return value in the caller. (As shown in Fig. 2.1, return variable "AA000001" is assigned to variable "t", which holds the actual return value for the caller.) Otherwise, the return value is ignored.

Davidson and Holler [5,6] showed several detailed steps for source-to-source inlining. However, they did not describe some important techniques, such as function-body relabeling, which will prohibit recursive inlining if not properly handled. Further, they failed to discuss the situations under which callsites could be inlined and the possible complications arising from attempting to inline recursive calls. This thesis addresses these issues in Chapter 4 by defining standard callsite formats and describing the process of callsite standardization.

2.3 Increasing Opportunities for Optimizations

After one inlining step, there might be some optimization opportunities created across former function boundaries because inlining overcomes the syntactical independence of functions. An example is given in Fig. 2.2. As shown in Fig. 2.2(b), due to inlining, there is a constant expression created ("AA00001 = 3 * 15 + 1"). This enables a compile-time evaluation that optimizes the constant expression into its corresponding value, 46, as shown in Fig. 2.2(c).

2.4 Considering Cache Effects

McFarling [12] did some valuable research on cache-performance considerations when making inlining decisions. He considered program structure (especially loop structures), cache-miss penalty, cache size and cache-replacement policies. Different cases, related to the placement of callsites within loops, were carefully studied and an inlining-decision algorithm based on cache issues was also given. After careful analysis of possible cache effects, we decided that our inlining-decision algorithm would only adopt a few basic ideas from this work, as explained in Chapter 5.

```
…
t = A(15); /* callsite */
…


int A(int i){
 return 3 * i + 1;
}
```

Fig. 2.2(a) program before inlining

Inlinin

```
…
{ int temp_i = 15;
   int AA00001;
  { AA00001 = 3 * 15 + 1;
     goto exit_01;
   }
  exit_01: t = AA00001;
 }
…
int a(int i){
  return 3 * i + 1;
}
```

Fig. 2.2(b) program after inlining

```
…
{ int temp_i = 15;
   int AA00001;
  { AA00001 = 46;
     goto exit_01;
   }
  exit_01: t = AA00001;
 }
…
int a(int i){
  return 3 * i + 1;  }
```

Fig. 2.2(c) program after inlining and constant-merging optimization

FIG. 2.2 INCREASED OPTIMIZATION OPPORTUNITY BY INLINING

2.5 Profiling to Guide Decisions

Several researchers [7,10,11,13] found that profile information helps to make better inlining decisions. The general idea of profile-guided inlining is to give the frequently executed callsites a higher priority when deciding what to inline, because they would lead to more instructions removed, hence shorter execution time and better efficiency. Most previous researchers made an assumption that the function callsites considered for inlining are not recursive. However, recursion is present in many practical programs.

Hwu and Chang [7,10] began to discuss recursion. However, they treated recursive functions the same way as non-recursive ones, except discussing some hazardous situations that might negatively affect the benefits of inlining (one such hazardous situation is seen in Fig. 2.3). The hazardous situation occurs if we inline a function *f* with a large number of local variables into a recursive function *g*. If the compiler allocates all local variables on the activation record, the large activation record might overflow the stack. Hwu and Chang concentrated more on inliner integration, profile-guided inlining and hazardous-situation detection. Many of their techniques have been successfully integrated into our greedy algorithm.

Scheifler [13] allowed recursion in his object-level inliner, but he assumed that there were no extremely large functions and he did not consider the hazardous situation of local stack explosion. An example of such an explosion is shown in Fig. 2.3.

```
…
A(i); /* callsite */
…
void A(int r){
 int p[1000000];
 /* body of function A */
A(j);
/* body of function A */
}
```

Local stack explosion →

```
…
{ int temp_r = i;
  int p[1000000];
  /* duplicated body of A */
  }
…
void A(int r){
 int [1000000];
/* body of function A */
A(j);
/* body of function A */
}
```

FIG. 2.3 EXAMPLE OF LOCAL-STACK EXPLOSION AFTER INLINING

As shown in the figure above, in case the inlined callsite is to a function that declares many bytes of local variables (allocated on the stack), these variables will be transformed to local variables with limited effective scope in the inlined code. Things will become worse if the inlined function happens to be recursive, since the stack size might increase greatly in the inlined program. This disastrous situation will occur if the compiler insists on allocating the array's space when it constructs the activation record, rather than on entering into the nested scope. It greatly increases the run-time cache misses, disrupts program locality, and may exhaust virtual-memory swap space. The benefit gained from inlining and the cost of increased stack use should be carefully balanced.

Kaser and Ramakrishnan [11] especially studied inlining of recursive functions and estimated the effect of one inlining step, extending Scheifler's work. Consideration was also given to the version issue (discussed in Section 5.4.5) and putting limits on code expansion. Detailed derivations of formulas to estimate the effects of a single inlining step were also given. Our greedy inlining-decision algorithm is primarily based on this paper.

In our implementation, the inliner needs accurate profile information to make better decisions. This is done by compiling the standardized program, running the executable, invoking the proper profiler and analyzing the profile data. Note that only callsite frequency and the number of times each function was entered are necessary, and thus collected. All other profile information is ignored.

2.6 Benefiting from Inlining

Inlining a callsite affects performance by reducing the number of instructions executed and increasing opportunities to optimize across function boundaries. As Davidson and Holler [5,6] as well as McFarling [12] discussed, the benefits fall into three categories.

- Removal of instructions

Every inlined callsite directly leads to the elimination of one pair of call-return instructions. This benefit can accumulate if a callsite inside a loop is inlined. Because the call and return instructions are usually expensive [10] and they are some of the most frequently executed ones. (Hwu and Chang [10] did some analysis and found that call and return accounted for about 4.5~6.5% of the instructions executed in some commercial programs.) The benefit gained from instruction removal is significant.

- Removal of loads and stores of parameters

When a callsite is eliminated, it is unnecessary to generate code to construct activation records, or pass parameters into the record or load and store different registers for parameters passing before calling the subprogram. The need to destroy the activation record before control returns is also removed. This benefit varies, depending on the number of parameters being passed in and the type of each parameter. Detailed timing analysis can be found in [6].

- Removal of Barriers to Optimization

Most of the optimization opportunities gained by inlining come from the possibility to optimize across former function boundaries. (Many compilers [8,19,20] do not attempt inter-function optimization, or do not do them well.) Constant propagation and its corresponding compile-time evaluation [17] will be introduced in Chapter 4 and are often considered very important.

2.7 Combining Techniques

A major special contribution of the thesis is a complete implementation of a multi-technique inlining-decision algorithm that is used to promote execution efficiency under acceptable code expansion. Some techniques have not been used together before, so it was important to see how they would interact. Some of the techniques that needed to be combined include

i). techniques to estimate the behavior of recursive function after inlining

ii). techniques to let the inliner choose between different versions of the callee

iii). techniques to collect profile information and guide inlining decisions

iv). techniques to predict cache behavior

v). techniques to implement constant propagation during inlining.

The combined technique tries to balance conflicts among individual techniques. One of the possible conflicts is given in Fig. 2.3.

As shown, different considerations lead to contradictory decisions about whether to inline callsites inside a loop. From the usual profile-based point of view [5,6], both callsites of A should definitely be inlined, because the inlined code would avoid many expensive call and return instructions.

```
…
 for(i=0;i<1000;i++){
    A (i);
    A (i);
  }
…
void A(int index){
 /* size of function A is about 80% of cache size */
}
```

FIG. 2.4 CONFLICTS IN COMBINED TECHNIQUES

However, consideration of cache performance [12] will not confirm this decision, because the expanded code will have many more cache misses. It remains for our greedy decision algorithm to combine these results and decide which callsites we should actually be inlined.

The decision algorithm tries to work automatically on the source program. None of our tested compilers[2] perform implicit inlining optimization, but a few do conduct inlining after the programmer's explicit specification[3]. However, all tested compilers give no control on the specific callsites that can be inlined. (A programmer can specify a function definition to be inlineable, so that the compiler tries to inline *every* callsite invoking the function). Our inliner will automatically consider each callsite in the source program without the programmer's specification and do the actual inlining after the algorithm has made its decisions.

---

[2] Borland C++ 5.02 for Win95/NT, Microsoft Visual C++ 5.00 for Win95/NT, Watcom C++ 10.0 for Win95 and Gnu g++ 2.7.2.1 for PC-Linux.

[3] Either on function definition or compiling switch.

The decision algorithm heavily uses profile information to guide its decisions. Our inliner will compile, execute and profile source code implicitly as part of the overall inlining system. Accurate profile information is then generated and directly used to form better inlining decisions.

Our overall inlining system must also convert programs from textual source code to an internal parse-tree form. Front-end processing – lexical analysis and parsing – is unavoidable. In the next chapter, we will introduce the design and implementation of our lexical analyzer, parser, parse-node class hierarchy and the various techniques used in the part of the overall system that does the front-end processing.

Chapter 3

Parsing Techniques

3.1 Introduction

Using a flex [1] generated lexical analyzer and a byacc [2] generated ANSI C parser, the C source program is converted into a parse-tree structure. The lexical analyzer (also called scanner) is used to partition the input stream into tokens and feed them to the parser, while a parser is built to accept a specified grammar and construct parse trees. Since all inlining techniques will later be applied on the parse tree, a good parse-tree data-structure design is relatively important.

The traditional way to build a parser is by writing the language's grammar in a form suitable for a parser-generator tool. Actions are designed to generate an in-memory abstract syntax tree and symbol tables during parsing. For efficiency reasons, a union is often used to record recognized syntactical constructs and merge different types of data. In our implementation, the parser was built by adding actions to construct our parse trees into an existing byacc-compatible ANSI C grammar [4]. This guarantees that the syntax is ANSI C standard and the most up-to-date grammar[4] is used.

---

[4] ANSI C standard was published in 1984 -- ANSI/NISO Z39.48-1984. A revision is available on ANSI/NISO Z39.48-1992. (National Information Standards Series, ISSN 1041-5653). Transaction Publisher, New Brunswick, 1993 (©1992). ISBN 0-88738-932-5.

The parser requires that proper parse-tree data be initialized in each recognized action. Several object-oriented design patterns are used to guide the parse-node design. We will first consider the class hierarchy that the design patterns lead to. Then the parse-tree details and three different dynamic arrays for parse trees used in the implementation will be introduced, followed by a discussion of the design patterns selected. At the end of the chapter, the message system, memory-management strategy, the standard interfaces and reparsing will be presented.

3.2 Class Design

The class design is to construct the class architecture in such a way that the byacc actions can instantiate an object of the appropriate subclass when the parser has recognized a syntactically valid structure. The entire class architecture is developed with the help of C++ design patterns, whose details will be discussed later.

3.2.1 Class Hierarchy

The complete class hierarchy contains 38 parse-node classes and is shown in Fig. 3.1, using the UML[5] notation. (By talking about "parse-node class", we refer those leaf classes in the class hierarchy that are derived from class *Node*. They are instantiated when building parse trees.) In the figure, there are three types of symbols: all leaf descendents of class *Node* represent general parse-tree nodes, all final descendents of class

[5] Unified Modeling Language. Details and standards can be found in [27].

DynamicNode represent non-terminal leaf-node classes and all final classes *BasicTypeNode* represent terminal leaf-node classes.

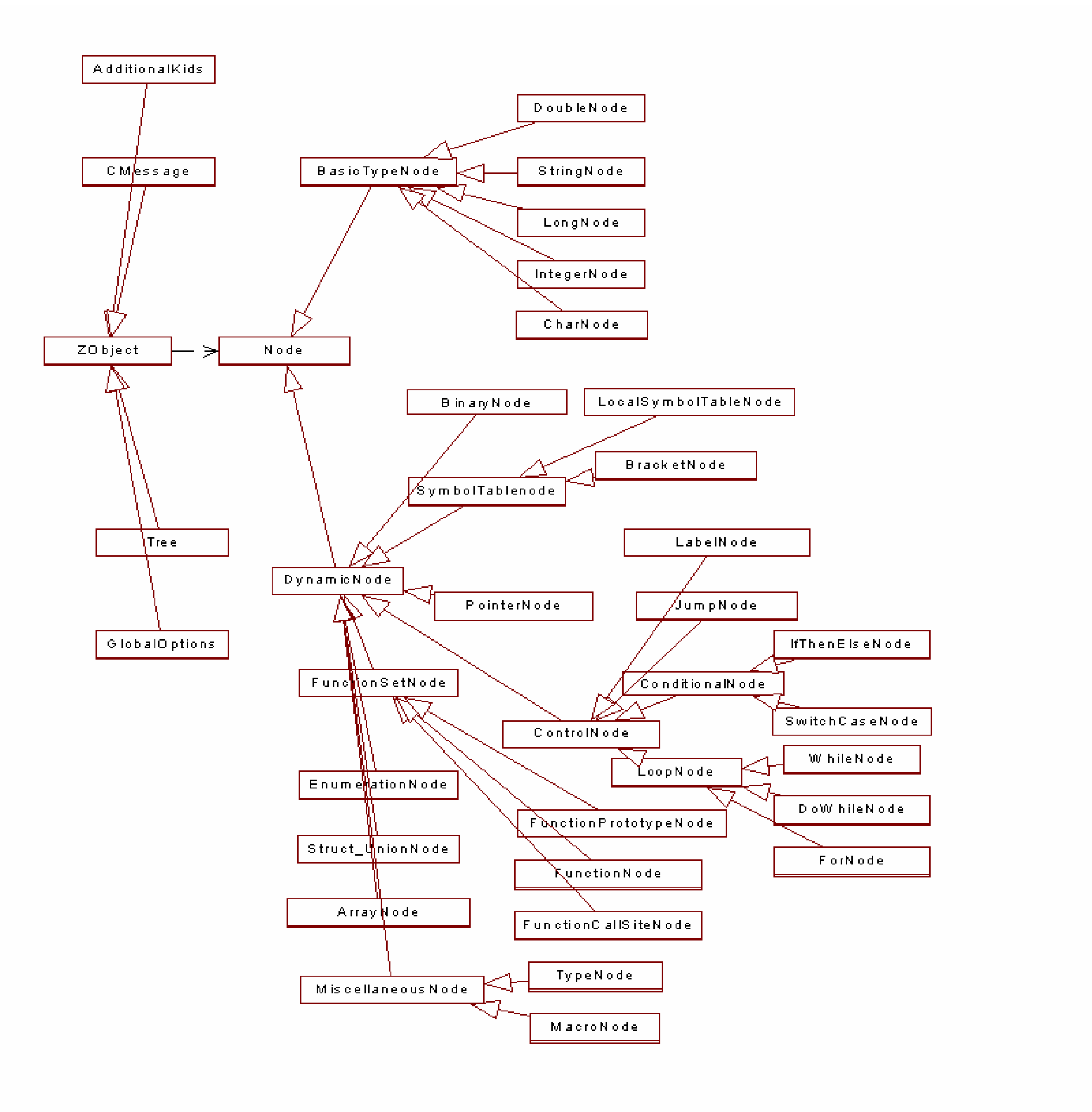

FIG. 3.1 COMPLETE CLASS HIERARCHY

In our design, the terms "terminal" and "non-terminal" indicate whether a specific class has internal subtree pointers and whether messages arriving need to be redirected. The "leaf" and "non-leaf" refer to positions in the inheritance hierarchy. Details about the message system will be discussed in Section 3.4.2.

Recall that the class hierarchy is developed under C++ design patterns and this leads to several desirable features that are achieved by this design. First, every C construct corresponds to *exactly* one class, which is a descendent of class Node. Second, only leaf classes are instantiated, all others are abstract classes. Then, only descendents of BasicTypeNode can be terminal nodes, which terminate message broadcasting. Furthermore, all classes are derived from one root class –ZObject – and thus form a single-inheritance tree. Finally, no multiple inheritance is used. This class design simplifies coding and debugging of the entire project. Two simple C fragments and their corresponding parse trees are shown in Fig. 3.2 and Fig. 3.3. The complete data-dependency relationship that shows the contents of each parse-node class is given in Appendix 1.

The C code in Fig. 3.2 has an ***if_then_else*** construct, where both ***then*** and ***else*** statements are not empty. This construct is directly mapped to an ***IfThenElse*** parse node, where the three subtrees (***condition***, ***then*** statement and ***else*** statement) are represented by Tree pointers in the ***IfThenElse*** node. Also, we see that every parse node has one pointer, its *self-containing* pointer, which points to its own Tree node. Details will be discussed later.

3.2.2 Symbol-Table Organization

The traditional compiler symbol table is an array-like structure, whose entries contain symbol type, name and possible initialization information. This inliner does not use this traditional design, since the inliner's symbol table is a dynamic array of pointers, where each entry points to the root of a local parse tree representing a local-variable declaration. Fig. 3.3 shows the parse-tree representation of a local symbol table. In every *BracketNode*, the first pointer is reserved for the local symbol table.

A limitation of this traditional symbol-table design is that the design becomes more complicated unless one assumes that each function has only one local symbol table and all variables need to be declared at the beginning of the function definition. This leads to some complications when local variables are declared in the beginning of a local scope (for example, variables may be declared following an arbitrary curly bracket). Our design arises because both the declarations and the statements are to be parsed and processed in the same way. Thus, it is easier to maintain and manage Tree pointers in symbol tables than analyze and record information in the conventional way.

```
…
if( x < 3)   x = 5;
       else  f(x-2);
…
```

Fig. 3.2(a) C source code for C Parse Tree

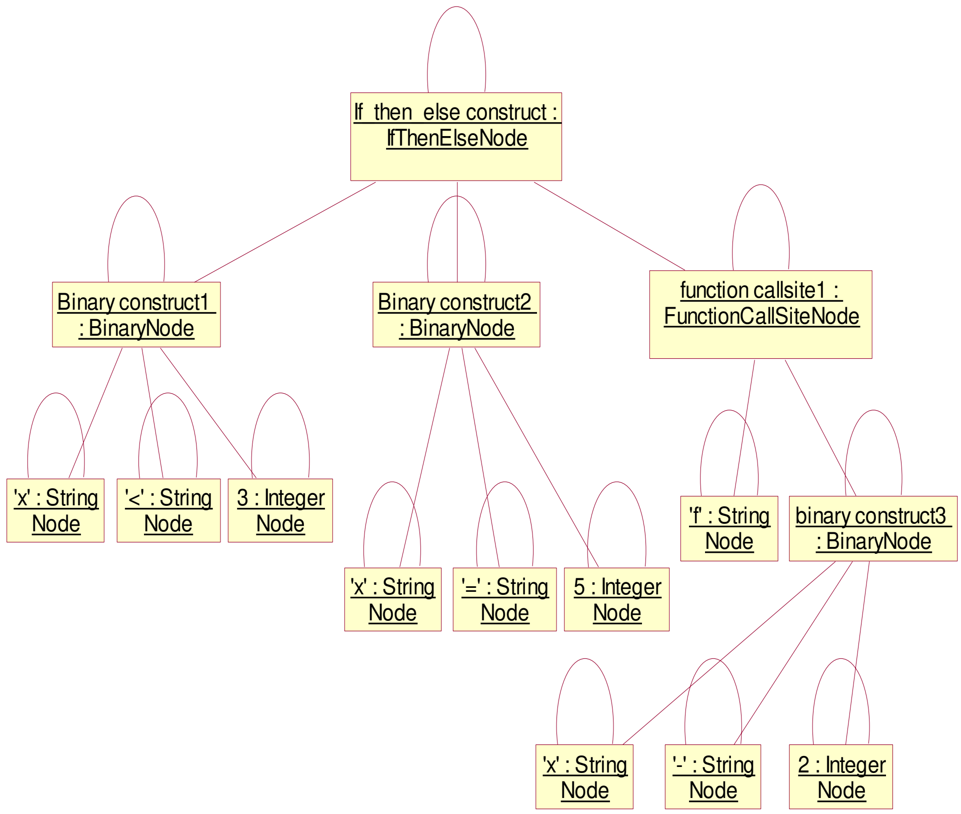

Fig. 3.2(b) C Parse Tree of a *if_then_else* construct

FIG. 3.2 EXAMPLE OF C PARSE TREE

3.3. Micro-Architecture and Message System

The class micro-architecture describes common features of the parse-node class design that are required for a message system. The message system requires that each parse-node class have the ability to process (process, propagate or ignore) messages. The class micro-architecture reflects these requirements and it supports the message channel that exists in each class and a set of class-specific message response routines that will actually do the parsing and inlining work. All parse-node classes are involved with the message system.

All parse-node classes have a similar micro architecture, which is shown in Fig. 3.4. ***NameOfClass*** identifies the specific class. ***Punctuation*** flag pairs include parenthesis, curly bracket, square bracket, comma, semi-colon and new line. They are used to write proper punctuations to the output stream to maintain valid C syntax. For instance, within the declaration ***int i,j;***, there is a comma after *'i'* and a semicolon after the current declaration. Only the comma flag of symbol *'i'* and the semicolon flag of the entire declaration are active. All others neither have active flags nor need to output anything. The proper punctuation flags are directly written into parse nodes when parsing.

Also note that by storing the punctuation flags, the parser constructs somewhat literal concrete parse trees (rather than abstract ones). Because the inliner will not do as much analysis as traditional C compilers do, it is relatively easy to store necessary punctuation flags and thus avoid having to regenerate them when restoring from the parse tree to a C source program.

```
{ int  i;
  long j; /* local symbols */
  i = 0;
…
}
```

Fig. 3.3(a) C source code with local symbol table

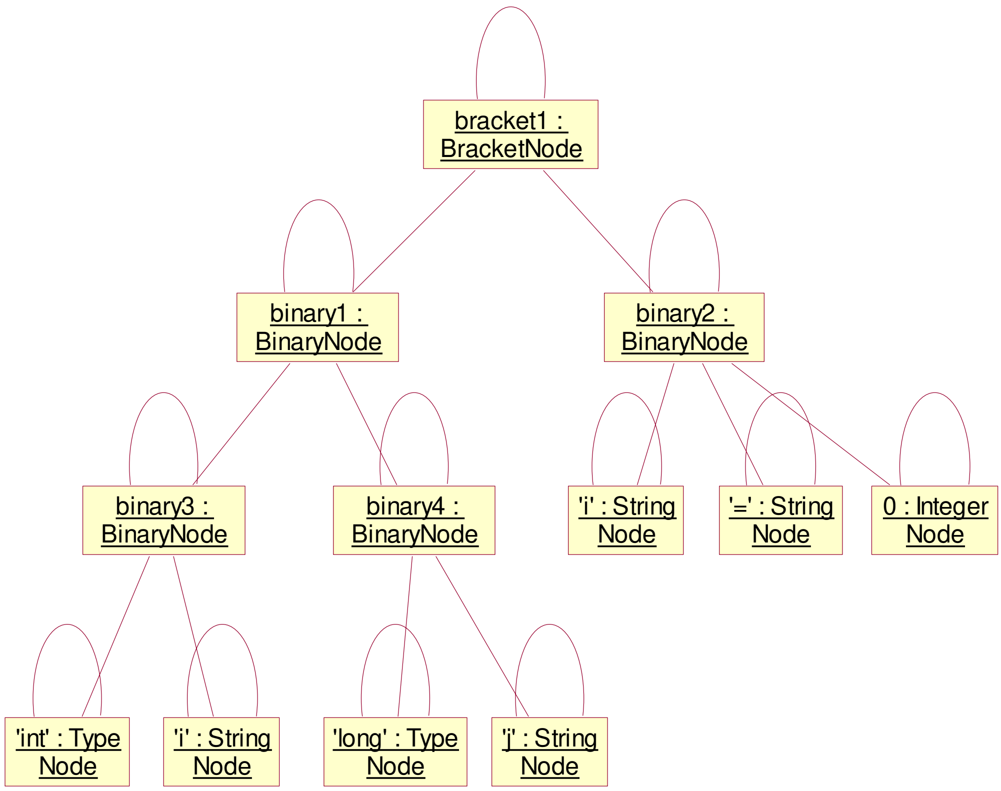

Fig 3.3(b) Example of Parse Tree Structure with Local Symbol Table

FIG. 3.3 EXAMPLE OF PARSE-TREE STRUCTURE WITH LOCAL SYMBOL TABLE

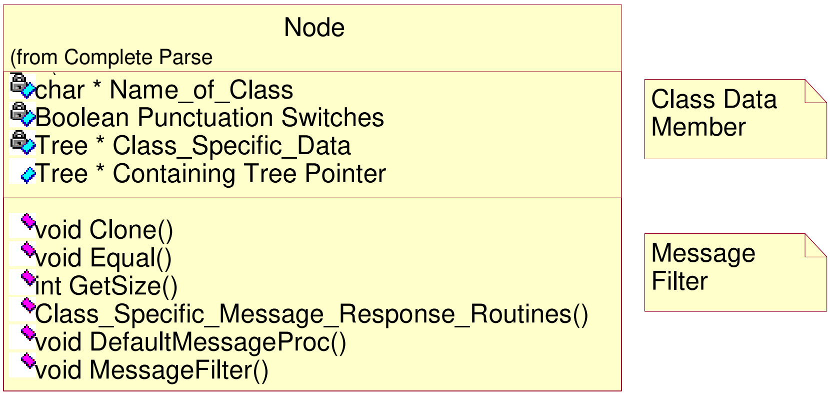

FIG. 3.4 PARSE-NODE MICRO-CLASS ARCHITECTURE

Class-specific information is, of course, different for each class. For an instance, a node of class ***IfThenElse*** has three Tree pointers that point to its *if* condition, *then* statement and *else* statement subtrees. However, the ***FunctionCallSite*** node has two tree pointers that point to the function name and its parameter list. The full description of parse-node classes is in Appendix 1. ***Containing-Tree pointer*** is used to reference the specific Tree object with which the node associates.

The Tree class is designed to have only *one* data element, a Node pointer. Each Tree object has a one-to-one relationship with a Node object, and all messages are broadcast through the Tree pointer and then redirected to its associated Node pointer. As well, memory sharing is achieved by using a reference counter that is a data member in class Node. Message broadcasting is described in Section 3.4.2 and memory-sharing details will be given in Section 3.5.

The actual work each parse-node class does is performed by the message filter that exists in each parse-node micro-architecture. It will process a message whenever it arrives. For every class, it supports the following four methods.

• Class-specific equal – check whether two objects contain exactly the same information. This recursively checks all subtree pointers to see whether they have exactly the same node information.

• Class-specific clone – explicitly clone an object without using memory sharing. The clone message is broadcast to the Node pointer and all its possible subtrees, which will duplicate the node itself and recursively clone every subtree of the node.

• Class-specific getSize – get the approximated parse-node size at run time. It is required to approximate function size when making inlining decision. Details will be given in Chapter 5.

• Default processing – use a default method to process messages that are irrelevant to the class. Since the inliner generates many messages when standardizing the parse tree (see Chapter 4 for detail) and performing inlining, it is relatively easy to activate only appropriate classes on a certain message and ask the other classes to ignore it. The message-system design makes the classes implicitly pass the messages to their internal subtrees instead of making the programmer write similar code several times. In this design, every parse-node class has default mechanisms to process messages, and the main message filter is used to check whether the incoming message is relevant to the class. If so, appropriate methods will be invoked to process it. Otherwise, the message will either stop or be recursively broadcast, depending on the type of the message and the specific parse-node class that is processing the message.

3.4 OO Design Techniques

Object-oriented systems have many built-in desirable features, such as reuse, extensibility, modularity and performance. Reuse of a successful class "architecture" is often achieved by using design patterns [3], which help to build a successful application in our particular domain – parsing and inlining. One of the key architectural components that the patterns enable in our application is the message system. After discussing the five applicable patterns used, the details of the message system will be given.

3.4.1 Design Patterns

Object-oriented design patterns are the solutions that experts use to solve problems. When they face a new problem, they usually recall a similar one that they have already successfully solved and reuse the essence of the solution instead of creating a complete distinct solution. There are five types of design patterns used in our inliner's implementation. We will discuss each in detail.

- the singleton pattern – create one object instance only. This is particularly useful for collecting and maintaining different versions of parse trees and it also is used for decision-making data structures.
- the builder pattern – incrementally build complicated classes. All parse-node classes are incrementally built by inheriting from class Node. The class hierarchy greatly simplifies the message-response routines.

• the facade pattern – provide simple and similar interfaces for subclasses. There is much similarity among the parse-node classes' declarations (e.g. their clone and equal messages), even though the implementation parts are completely different.

• the proxy pattern – let one object simulate another. In our inliner, every parse node can freely be used to dynamically simulate any other parse node once at inlining time. This is achieved by maintaining a redundant Tree pointer in each parse-node object and redirecting all class behaviors to this redundant Tree pointer whenever the object is marked "dirty". For example, the FunctionCallSiteNode can be used to simulate the function body, and this provides the kernel of the inliner. Details will be introduced in Chapter 4.

• the bridge pattern – provide a message-broadcasting mechanism. All messages are broadcast to a Tree pointer and propagate to its associated Node's subtrees. The general idea of the Bridge pattern is to support the construction of an abstraction hierarchy (parse-node class definitions) parallel to implementation hierarchy (parse-node class implementations) and to avoid permanent binding between the abstractions and the implementations, and hence to reduce dependency among software components.[6] The bridge pattern is used to connect the Tree pointer with all kinds of parse-node classes. The actual class to which the bridge connects will be determined at run time. Thus, each parse-node class is required to have similar interface with the Tree bridge. The class-dependent distinct implementations successfully separate the permanent bindings

[6] In our implementation, every necessary action is passed through Tree bridge and the class methods involved are fully determined by message – no class method will be directly invoked at any time.

between Tree pointer and the parse-node, and it is easy to transparently change parse-node implementations without modifying the parser.

The Bridge pattern is implemented by the design of the Tree class (discussed in Section 3.4). Every Tree node has a Node pointer and every possible message broadcaster (a method in Tree class that is used to send out a specific message). The Node pointer is instantiated to a certain parse-node object, and so a message could be broadcast to the specific parse-tree object by sending the message to its Containing-Tree pointer. The message broadcasters are used to send different messages to the Node's Containing-Tree pointer. Every message conceptually has one message broadcaster and the Tree class encapsulates the details of message broadcasting. As an example, a parse tree is presented in Fig. 3.5 with UML notation for class representation. We see the parse tree for the C statement ***t = 2***. There are four Tree pointers in this figure, each associates with a Node pointer instantiated to a specific parse-node: BinaryNode, StringNode, StringNode and IntegerNode, respectively. When a message is sent to r, the root of a subtree (e.g., the message to do a local replacement of string "i" with "local_i"), it is broadcast to every node in the parse tree beneath r. Only the interested nodes (the StringNode whose data name is ***i***) will process it. Others either propagate (non-terminal nodes) the message or simply ignore the message (terminal nodes).

3.4.2 Message System

While the current design uses a message system, the original design used a completely different one. Before discussing the message system, we should examine the original design and show the reason why it is desirable to introduce the message system.

The original design extensively used virtual methods as follows. Before a new method can be inserted into a leaf-node class, the same virtual method would have been placed into the Node class and all other parse-node classes, which do nothing but pass the message to their subtrees. This involves a lot of redundant code and an example is shown in Fig. 3.6.

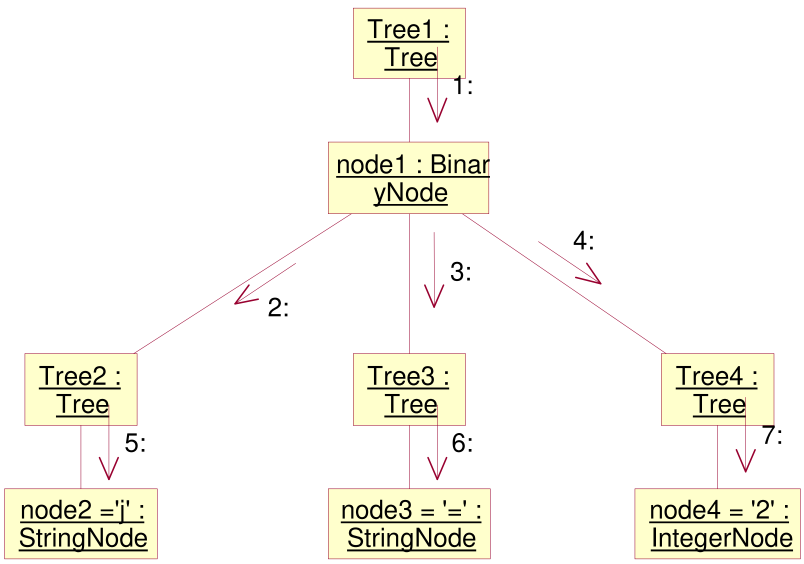

FIG. 3.5 EXAMPLE OF MESSAGE BROADCASTING SYSTEM

In the example shown in Fig. 3.6, a *GetName( )* method is being added to get the actual variable name from an array variable, a pointer variable or a string variable. A pure virtual method with the same name is inserted into the Node class first. This guarantees that the required method could be obtained directly through the Node class. Then different implementations of the same method have to be inserted into each relevant parse-node class. Even worse, sometimes the programmer would have to rewrite the method in irrelevant parse-node classes, just to propagate the message.

The message system was developed to avoid these difficulties. The message class is a data structure that contains message type, id, parameter and return value and an example message structure is shown in Fig. 3.7. The necessary message is generated and broadcast to the parse-tree nodes through their Containing-Tree pointers.

A message broadcaster is a method in the Tree class, which generates the message, broadcasts it out and takes the return values if necessary. It is different from the parse-node message-response routines, which actually do the work the message system desires. Message and message broadcaster have a one-to-one relationship, while message and message-response routines have a one-to-many relationship.

The message life cycle consists of message generation, broadcasting/propagating/processing and termination. First, a message is generated when performing a specific function whose data or result cannot be obtained directly from Tree pointers at parsing or inlining time, as shown in Fig. 3.7. This is done by constructing a CMessage structure. Then, the message is broadcast to the parse tree through the message channel, by passing the CMessage structure as the parameter in all message broadcasters and message-response routines. The message is initially directed to a Tree pointer, which could be any node in the current parse tree, and then is usually propagated top-down to the entire local parse tree. The message propagation terminates when the message reaches a terminal node, where it is either processed or ignored. Note that every subtree could serve as a place where a message broadcast could begin.

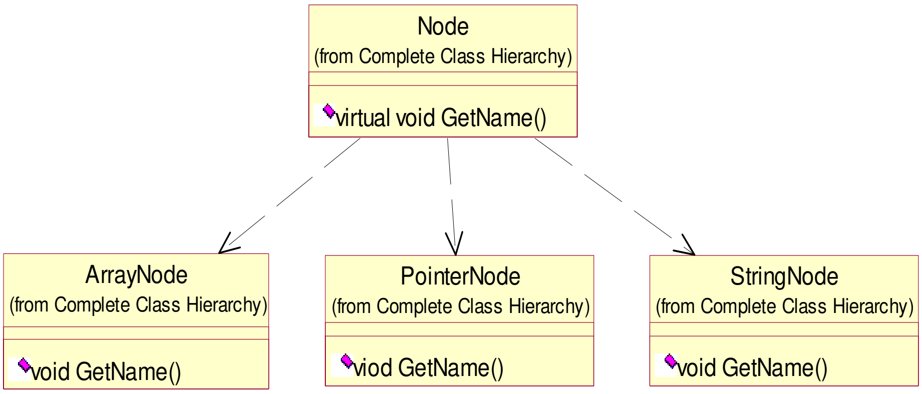

Fig 3.6(a) Class Hierarchy for Virtual Method Mapping

```
Tree * ArrayNode::
GetName( ){
 return
ArrayName->GetName( );
}
```

```
Tree * PointerNode::
GetName( ){
  if(not_nested_pointer)
  return PointerName;
else
  return
PointerName->GetName( );
}
```

```
Tree * StringNode::
GetName( ){
   return StringName;
}
```

Fig 3.6(b) Pseudo-code for Virtual Method Mapping

FIG. 3.6 EXAMPLE OF VIRTUAL METHOD MAPPING

```
func_declaration: fun_type fun_name '(' param_list ')'  fun_body {
     Tree * nowfun = new tree("funcnode");
     nowfun->setfunreturntype($1);
     nowfun->setfunname($2);
     nowfun->setfunparam($4);
     nowfun->setfunbody($6);
     $$ = nowfun;
  }
```

FIG. 3.7 EXAMPLE SHOWING MESSAGE BROADCASTING IN PARSING

The example in Fig. 3.7 shows a situation where a series of messages are generated and broadcast when the parser recognizes a function definition. Now that the parser has function return type, name, parameter list and the function body's parse-tree pointer in variable $1, $2, $4 and $6 respectively, it is necessary to record them into a Tree pointer, *nowfun*, which has instantiated a FunctionCallSiteNode at the place of its Node pointer. However, *nowfun* is just a Tree pointer and it has no idea that its Node pointer contains a FunctionNode. A series of messages are generated and broadcast to *nowfun* Tree pointer, to properly record the necessary information. This is also a case where a parse-node class will only be interested in relevant messages. For example, if the same sequence of messages is broadcast to a Tree pointer that contains a string value, this parse node will simply ignore any message it receives in this case and no information will be recorded.

```
expr_stmt: expression operator expression {
  Tree * Result;
  Result = new Tree("binnode");
  Result->SetBinLeft($1);
  Result->SetBinOperator($2);
  Result->SetBinRight($3);
  $$ = Result;
  }
```

FIG. 3.8 EXAMPLE OF MESSAGE BROADCASTING IN PARSING

Another parsing example is given in Fig. 3.8, which shows what happens when the parser recognizes a binary operation and generates a sequence of messages that are used to record the left, operation and right subtrees. It is easy to see that the message sequence in BinaryNode recognition is completely different from that used to record

FunctionNode information. However, if this sequence of messages is broadcast to a Tree pointer that has a function callsite instantiated, all messages will also be ignored. Note that every parse-node class has a different set of methods that is only related to its specific class. Even methods with the same name but located in different classes will have different implementations and behaviors.

3.4.3 Message Composition

The composition of the CMessage class is presented in Fig. 3.9, showing all the necessary data this class needs to have. This includes

• Message Type – identifies whether the message should be broadcast up or down in the parse tree. Most of the time, the message needs to go down in the parse tree, in the techniques required for inlining. In special cases, for instance, to get the function return type from within the current function body, the message will go up in the parse tree.

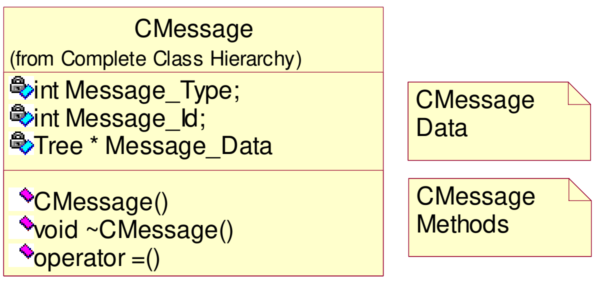

FIG. 3.9 CMESSAGE

• Message Id – indicates the message type. Every message has a predefined integer value for its Id. By checking the message Id in the class message filter, a class realizes immediately whether this message is relevant and thus whether it should process the message or invoke the default message processor.

• Message Data – contains the incoming parameters and the outgoing return values. This includes a set of variables for each basic type and three Tree pointers, because different messages could return different types and numbers of results. Our experience with the inliner is that this message data is sufficient.

3.4.4 Message-Design Evaluation

Fig. 3.10 presents three methods in pseudo-code: a general message broadcaster, a message processor and the message connector that ties the Tree pointer with its associate parse-node data. The message is built by constructing a CMessage, which is then broadcast to the appropriate Tree pointer. Then the Tree pointer redirects the message to its current parse node. Note that whenever a message is broadcast, it must start at a Tree pointer, so that message propagation could begin. The response routines in the parse node then process a message whenever it reaches a parse node. The result is passed back in the CMessage structure and returned from the message broadcaster. Every parse-node class has a default message processor, which captures the messages and checks its message id. All irrelevant messages are processed by the default mechanism, which does nothing but either propagate or ignore them. The default message filter invokes the appropriate method to process the message if it determines the message is relevant to the specific class.

There are some advantages to the message system. First, it provides a standard interface for message response routines. Their standard format is: void ClassName::RoutineName(CMessage &). Second, it passes messages implicitly. Every parse node has a default message-processing routine called *DefMessageProc*, which is responsible for redirecting all the irrelevant messages to its subtrees. This is different for every class, as presented in Fig. 3.10.

The design goal of the message system is to reduce the programming complexity and avoid code explosion. Under this design, it is unnecessary to specify the processing of irrelevant messages. All irrelevant messages will be processed by the class's default method, which passes messages implicitly to the class-specific subtrees.

```
Tree * Tree::
HandlerName(Parameters){
    Tree * Result = NULL;
    CMessage Msg;
    Msg.Id    = Identification;
    Msg.Type = Type_of_message;
    Msg.Param = Parameters;
    BroadCastMessage(Msg);
    Result = Msg.Result;
    Msg.Clean();
    return Result; }
```

3.10(a) A general message broadcaster

```
void
ParseNode::BroadCastMessage(CMessage
& Msg){
  switch(Msg.Id){
         …
  case Type_of_message:
     ProcMsg(Msg);
     break;
      …
   default:
      DefMessageProc(Msg);
      break;  }
  }
```

3.10(b) A general message processor

```
void Tree:::BroadCastMessage(CMessage  &Msg){
     if(node_ != NULL)
        node_->BroadcastMessage (Msg);
   }
```

3.10(c) Tree to parse-node message connector

FIG. 3.10 EXAMPLE OF MESSAGE-BROADCASTING SYSTEM

However, there is an apparent disadvantage of the message-system design – the large number of messages. There are currently about 200 messages in the current implementation (Appendix 4 shows the most important 60 of them). Theoretically speaking, each message will have a message broadcaster, which makes the Tree class very *fat* in that it has many methods. Some tricks are used to shrink the number of message broadcasters to around 13.

```
void BinaryNode::
DefMessageProc(CMessage
&msg){
if(msg.Type == SYS_UP)
Parent->BroadCastMessage(msg);
else {
  if(Left != NULL)
   Left>BroadcastMsg(msg);
 if(Operation != NULL)
  Operation->BroadcastMsg(msg);
 if(Right != NULL)
   Right->BroadcastMsg(msg);
    }
}
```

Fig 3.11(a) Default message processor for BinaryNode

```
void FunctionNode::
DefMessageProc(CMessage
&msg){
if(msg.Type == SYS_UP)
 Parent->BroadCastMessage (msg);
else {
 if(FName != NULL)
 FName->BroadcastMsg(msg);
 if(Param != NULL)
 Param->BroadcastMsg(msg);
        }
}
```

Fig 3.11(b) Default message processor for Function Node

FIG. 3.11 EXAMPLE FOR DEFAULT MESSAGE PROCESSOR

3.5 Memory-Management Strategy – Sharing + Cloning

Since C++ does not provide automatic memory management and the object-oriented parser has high memory consumption, memory management has been carefully considered for parsing and inlining. The smallest parse node is IntegerNode, whose size is 44 bytes. On the opposite extreme, one FunctionCallSiteNode requires 130 bytes.

Because of the *big* node problem caused by parse nodes, we will discuss memory sharing that reduces the high memory consumption without harming the advantages we achieved from the design.

3.5.1 Memory Sharing

The memory management is controlled by ***NEW*** and ***delete*** operators where C++ will allocate and free memory to a specific structure. However, C++ has an implicit limitation on overloading the memory allocation operator ***NEW***.

As shown in Fig. 3.12, the overloaded C++ ***NEW*** operator must have a size_t parameter first. This makes it impossible for an overloaded **NEW** operator to accept the Tree pointer as its first and only parameter.

```
void * operator new (size_t size, …){
 /* actual work overloaded new will do */
}
```

FIG. 3.12 OPERATOR ***NEW*** OVERLOADING

Due to this limitation, a **NEW** operator was created and explicitly used for Tree pointer sharing. By introducing a reference counter and overloading the memory management operators, a relatively easy strategy was developed to make efficient use of memory – sharing. Both ***NEW*** and ***delete*** operators are overloaded to make the implicit memory sharing possible. An example is shown in Fig. 3.13.

Because all the parse node memory operations are manipulated directly through Tree pointers, it is sufficient enough to overload memory management operators (***NEW*** and

delete) for the Tree class only. However, in some special cases (e.g. duplication of the inlined function's body), shared copies cannot be used and explicit cloning is unavoidable.

```
void Tree * NEW (Tree * InitTree){
 if (InitTree) {
   InitTree->node_->ReferenceCounter++;
   return InitTree;  }
 else   return NULL;
}
```

Fig 3.13(a) Example of ***NEW*** operator overloading

```
void operator delete (void *p){
  if (--((Tree *)p)->node_
        ->ReferenceCounter)
       return;
   else   ::operator delete (p);
 }
```

Fig 3.13(b) Example of ***delete*** operator overloading

FIG. 3.13 MEMORY MANAGEMENT OPERATOR OVERLOADING

3.5.2 Cloning

Memory cloning is supported as one of the four necessary methods in each class's message response routines. Every parse node has a *clone* method to duplicate itself and all its subtrees, and this method is implemented by explicit recursive duplication of the memory's contents. An example is presented in Fig. 3.14.

Because the parser does not implicitly perform memory sharing (exactly duplicates of constructs in the source code will generate different parse trees, rather then a shared copy), all sharing is created in the function-callsite analysis when making inlining decisions. The ReferenceCounter is not cloned when performing the function duplication. Details on the use of sharing with the call graph will be given in Chapter 5.

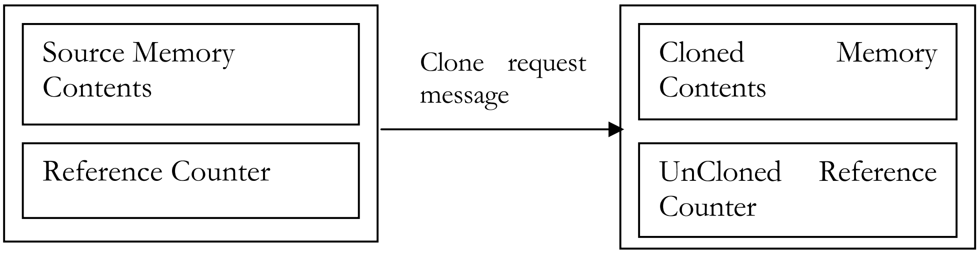

FIG. 3.14 EXAMPLE OF MEMORY CLONING

3.6 Parse-Tree Structure

Recall that a parser usually reads valid syntactical structures and constructs the parse tree to enable the code generation phase. Although we are not going to generate intermediate code, we still need this internal syntax representation to do our inlining work. The following will give some examples to show the way the inliner internally represents and manipulates the source program after parsing.

3.6.1 Collections of Parse Trees

From a global point of view, the entire "parse tree" is actually a disconnected forest, with three trees.

- GST List – Global Symbol Table List. The GST List contains all global-level declarations, such as constants, line directives, variables, arrays, struct and unions, enumerations, etc. Fig. 3.15 shows several simple global declarations and the corresponding GST List.

All global declarations are collected into the GST structure, which functions like a dynamic array of Tree pointers. The array is ordered the same way as the parser generates it – a declaration that appears early when parsing will have an early index in the dynamic array.

```
…
int i;
long j;
#line "gram.y" 333
…
```

Fig 3.15(a) C code with global declarations

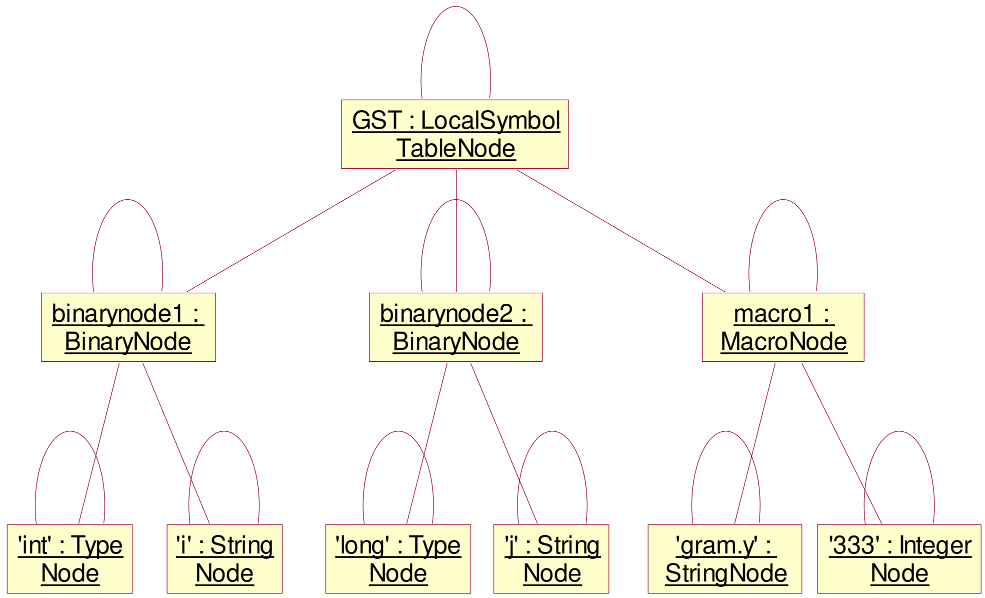

Fig 3.15(b) Example of Global Symbol Table for C code shown in Fig 3.15(a)

FIG. 3.15 EXAMPLE OF GLOBAL SYMBOL TABLE

• Function-Prototype List – This contains all declared function prototypes. As required by ANSI C, every function needs a prototype declaration before either the function definition or a callsite may appear.

Note that it is possible to merge the Function-Prototype List into the GST List since we can consider that function prototype is a kind of global declaration and all global declarations should be maintained the same way. However, it seems better to maintain the global function-prototype declarations in a different list, for reasons that will be introduced.

• Function List – this collects all user-defined functions.

All three parse-tree lists are arranged and managed in the same way and are similar data structures – dynamic arrays of Tree pointers. Note that the source-code line number is not kept in any of the parse-tree nodes except the "#line" directives whose specified line number must be recorded.

3.6.2. Reparsing

Reparsing refers to writing part or all of the current parse tree into source-code files, destroying the parse tree and regenerating it from the files. It is chosen to keep the entire parse tree consistent and always updated and it improves robustness and simplicity at the cost of efficiency.

There are two types of reparsing – global reparsing and local reparsing. Global reparsing will destroy the entire parse tree and regenerate everything. This usually will be applied after one inlining step, because after inlining once, one carefully chosen callsite will have been inlined. Thus the parse-tree structure has been changed, new call sites might have been created and updating the entire parse tree will have become necessary.

However, global reparsing is not always necessary. After each of the internal swapping and splitting (details in Chapter 4) operations, only a small portion of parse tree needs to be regenerated. This motivates local reparsing, which only updates a necessary subset of the entire parse tree (usually a function or an expression). The reparsing steps are the same.

3.7. Miscellaneous Parsing Techniques

Besides the parsing techniques introduced above, there are many minor ones used in parser design. A few will be examined that lead to a better overall understanding of the inliner.

3.7.1 Swapping and Splitting

A certain number of "swapping" and "splitting" operations will be applied on the parse tree to standardize function callsites (details will be given in Chapter 4). When carrying out one of these operations, we will scan the entire parse tree once, searching for occurrence of a pattern to be replaced with some other patterns. The way we rewrite the *if_then_else* construct is illustrated in Fig. 3.16(a), the abstraction of parse-tree representation and the pseudo-code description are given in Fig. 3.16(b), Fig. 3.16(c), Fig. 3.16(d) and Fig. 3.16(e).

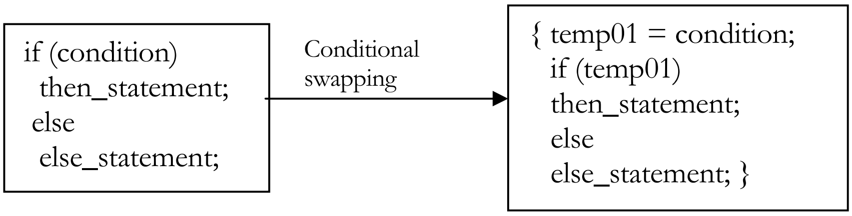

Fig 3.16(a) General rule for *if_then_else* construct swapping

```
void GlobalOptions::SwapAndSplit(void){
 …
  Root->SendMessage(ck_SwapIfCond);
/* Root is the root of the entire parse tree */
…
/* Other messages to standardize parse tree */
}
```

Fig 3.16(b) Pseudo-code for *if_then_else* construct swapping

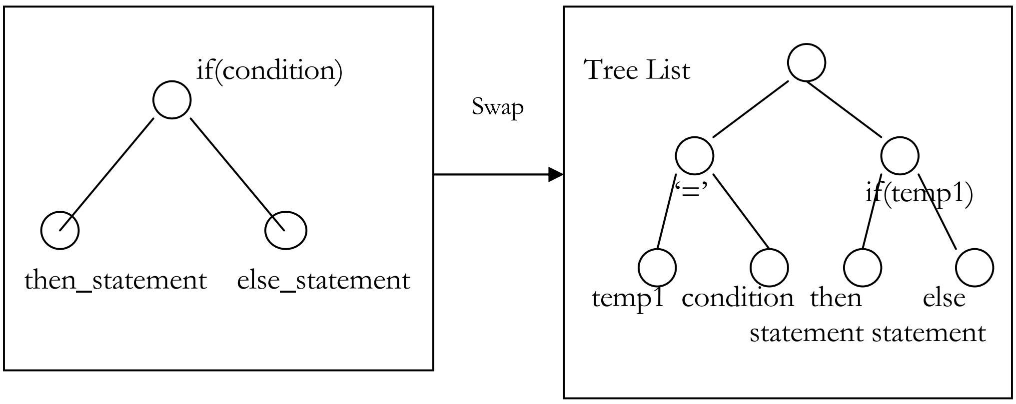

Fig 3.16(c) Compressed parse-tree representation for *if_then_else* construct swapping

```
void IfThenElseNode::BroadCastMessage(CMessage &Msg){
        Switch(Msg.Id){
         …
        case ck_SwapIfCond:
                SwapIfCond(Msg);
                break;
              … } // end of switch
  … } // end of message filter
```

Fig 3.16(d) Pseudo-code for *if_then_else* message filter in IfThenElse class

```
void IfThenElseNode::SwapIfCond(CMessage &Msg){
 /* Code to do the real if_then_else conditional construct
swapping */
 }
```

3.16(e) Pseudo-code for if_then_else conditional swapping

FIG. 3.16 EXAMPLE FOR *IF_THEN_ELSE* CONSTRUCT SWAPPING

Note that there is exactly one message for each swapping/splitting process and one swapping/splitting step will finish before the message returns. All message details could be found in Appendix 4. In each standardization process, a swapping/splitting message is sent directly to the parse tree root and propagated to the entire parse tree. Every C syntactical construct will receive this message, but only those that fall into that specific swapping/splitting category (in this case, the IfThenElseNode) will perform the actual standardization.

3.7.2 Standard Interfaces

Providing standard interfaces is another direct advantage of the chosen design patterns. This separates the parse-node implementation details from its definitions, keeps the parser concise and enables it to process large number of different messages. Because every action is performed through the message system, there is no need to invoke parse-node methods directly from the byacc actions. The parser will just send a message or a sequence of messages to a local parse-tree root, and the message system will connect to the appropriate message response routines and take care of the rest. This enables the transparent modifications and transplantation[7] of the parse-node implementation without modifying the parser. Two examples are given in Fig. 3.17(a) and Fig. 3.17(b).

The first is the interface between the parser and parse-node data structure. It shows the way the parser allocates a Tree pointer and connects an appropriate parse-node instance to it. This is used to keep the parser skeleton simple, maintaining the understandability during the project's development. This code is class independent and could appear anywhere in the parser. Note that the string "parse-node-type" represents the name of a parse-node class that needs to be instantiated. All parse-node names and data structure details are listed in Appendix 1.

The second standard interface is between the Tree class and all the parse-node classes, and is used when instantiating a Tree pointer. It ensures that every Tree node connects to only one appropriate parse-node. All the messages broadcast to this Tree pointer will be automatically redirected to its associated parse-node.

---

[7] A parse-node class could be replaced by another parse-node class without modifying the parser and message system by just redirecting the message to the new parse-node class.

```
…
Tree * DataName;
DataName = new Tree("parse-node-type");
…
```

Fig 3.17(a) Interface between parser and parse-node classes

```
Tree * Tree::Tree(char  * parse-node-type){
     switch (parse-node-type){
     …
     node = new ParseNodeName(this);
     … }
    return this;
}
```

Fig 3.17(b) Interface between Tree class and Parse-node classes

FIG. 3.17 EXAMPLE OF STANDARD INTERFACE

3.7.3 Parse-Tree Adjustment

The parser generates the local symbol table as defined by the ANSI C grammar and an example is given in Fig. 3.18. However, this kind of deeply nested BinaryNode parse tree is not suitable for inlining processing, especially when the inliner wants to take out one particular declaration and process it (e.g. renaming). Thus, the parser performs an internal scan, which converts all the deeply nested BinaryNode declarations to a flat array-like width-first declaration. An example is also given in Fig. 3.19.

```
… {int i;
    long f;
    char c; …}
```

Fig 3.18(a) C source code for local symbol table

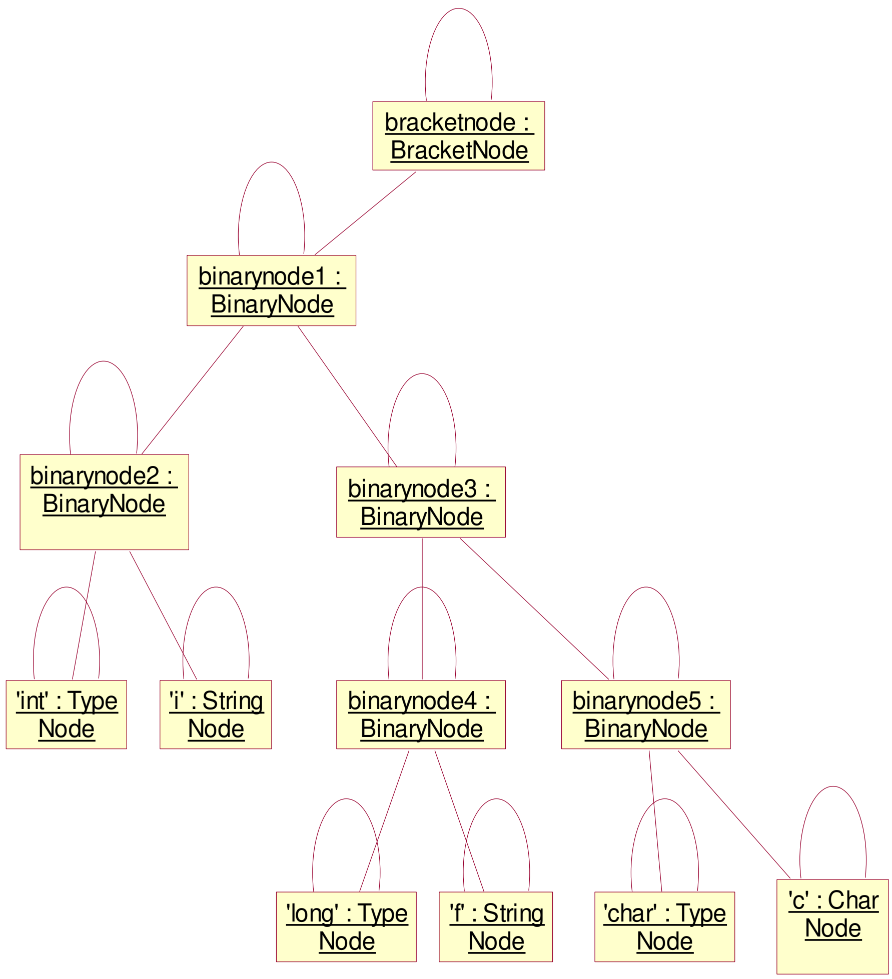

FIG. 3.18 EXAMPLE OF PARSER-GENERATED LOCAL SYMBOL TABLE

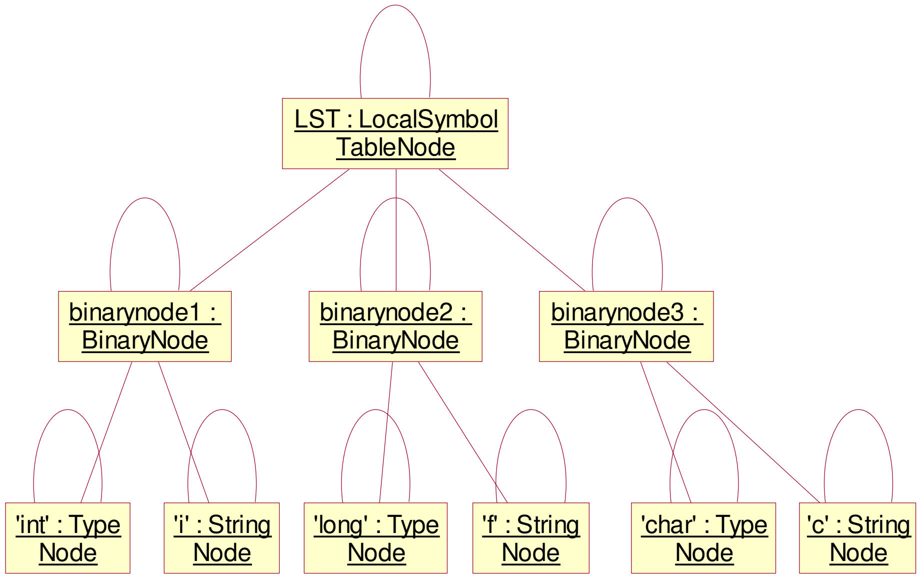

FIG. 3.19 IDEAL LOCAL SYMBOL TABLE

Chapter 4

Inlining Techniques

4.1 Top View

The inlining technique has two parts: standardization of each function callsite and inlining of candidates that have been selected by the decision algorithm. The top view of the entire processing is shown in Fig. 4.1 and consists of eight processing steps.

1. C Preprocessing

The C preprocessor is invoked at the very beginning, so that all macros are expanded and `#include` directives are processed. Every invocation that appears to be a function call, but is actually a macro invocation, is uncovered. This process generates a temporary file that is ready to be parsed, and the original file is unchanged.

2. Parsing

The second step is to parse the temporary file, using the lexical analyzer and parser introduced in Chapter 3, to construct the parse tree. Every valid C construct is recorded in its corresponding parse-node. All line directives ("#line xxxx") are recognized and recorded as global or local variable declarations in the parse tree.

3. Swapping and Splitting

Then the parse-tree information is fed to the swapping and splitting module in the third step. This scans the entire parse tree, rewriting and reparsing it several times. All function callsites are converted into a standard format that will be discussed in Section 4.2.

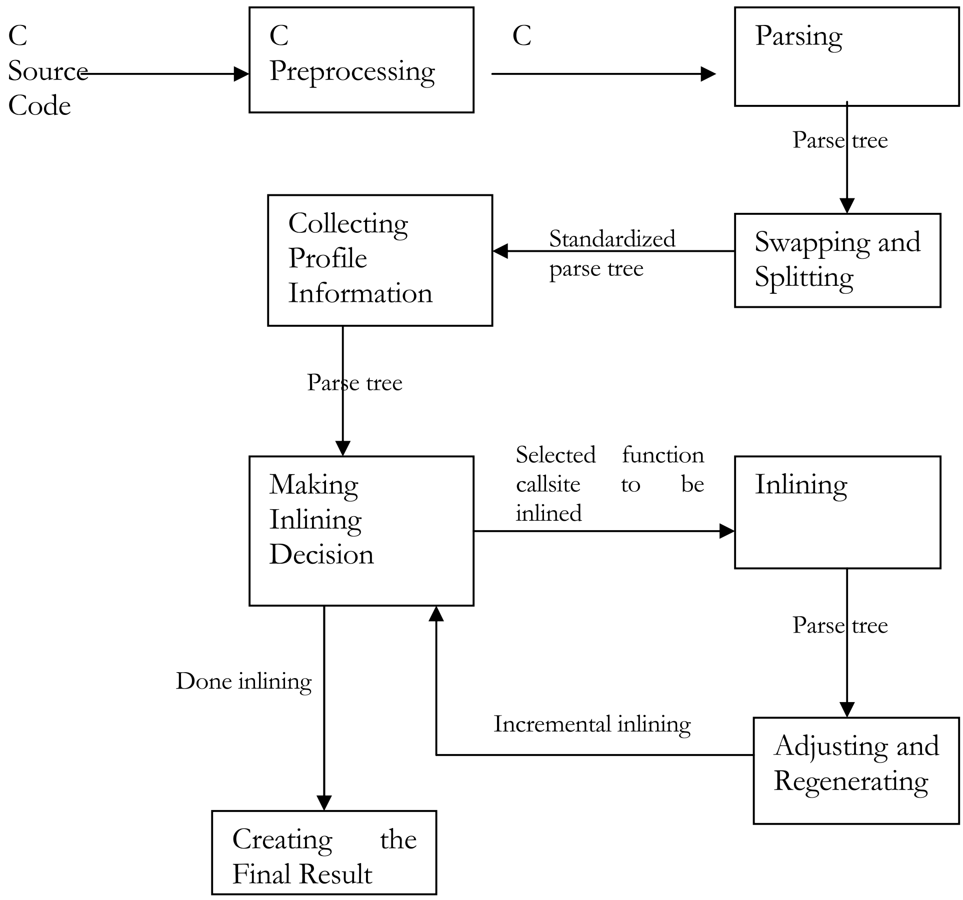

FIG. 4.1 ENTIRE INLINE PROCESSING

4. Collecting Profile Information

In the fourth step, profile information needs to be collected for the standardized parse tree. A modified version of the standardized parse tree is generated by cloning the original one and creating dummy functions at each callsite. It is then used to generate C source code that is compiled, executed and profiled. Accurate callsite frequencies and the number of times each function is entered are recorded and used for inlining decisions.

5. Making Inlining Decisions

The fifth step is one of the most important parts of the thesis work, since it makes the inlining decisions about which callsites should be inlined. To do this, all uninlineable callsites are removed from the list of inlineable candidate and routine sizes are approximated. Some complicated analyses are based on version issues and limited code expansion, and will be introduced in Chapter 5.

6. Inlining

Function inlining substitutes the selected callsite with the corresponding function body in the sixth step. Details involve duplicating of function body, simulating proper parameter passing, removing the callsite, rewriting labels and eliminating return statements. As well, identifier conflicts must be solved. These issues were discussed in Chapter 2 and will be expanded in Section 4.7.

7. Adjusting and Regenerating

After one inlining step, new callsites might have been created that could make the parse tree become inaccurate. Rather than incrementally adjust all data structures, we choose to reparse the inlined program or function in the seventh step. Re-profiling may also be done. This has already been discussed in Section 3.6.2, and further details will be given in Section 4.8.

8. Creating the Final Result

When the decision algorithm finds that there is no inlineable callsite worth inlining, we do the final step, the clean-up stage. The inlined C source code is generated from the parse tree into a file. Then the parse-tree structure is destroyed and the entire inlining process terminates.

4.2. Standardizing Callsite

Davidson and Holler [5] provided details of techniques for source-to-source inlining, but they neither showed the way to inline deeply nested callsites nor discussed the situations under which source-to-source inlining techniques could be applied to callsites. In this thesis, two standard callsite formats are developed to address this issue. Each format is a statement, and thus no rewritten callsites can be nested inside an expression. All inlineable function callsites are to be rewritten to one of the following standard formats and this greatly simplifies later inlining.

The two standard formats are

1. f(x);

f is the function name and ***x*** is a parameter list that does not contain any inlineable function callsites. This is the standard format for a callsite invoking a function whose return type is void or whose non-void return value is simply ignored.

2. y = f(x);

y is a simple variable name, **‘=”** is one of the assignment[8] operators, ***f*** is the function name and ***x*** is a parameter list that does not have any function callsites. This is the standard format for all function calls whose return results have to be recorded.

Callsites can appear in various forms in the source code and cannot be used for inlining before they have been converted into the standard formats, as introduced in Chapter 2. Swapping and splitting are the methods used to standardize each callsite, and they are introduced next.

4.3 Swapping

The swapping process lifts all function callsites out from within conditional control expressions, loop control constructs, declarations and return statements. For each such site, it creates a temporary variable, assigns the variable with the callsite swapped out and puts a copy of the variable’s name in the callsite’s original place. Details are now given to show how we swap various C constructs without changing the program’s semantics.

4.3.1 Swapping Conditional Controls

Since function callsites within control expressions cannot be inlined directly, swapping is used to process these sites first. There are five types of statements to consider. They are ***if_then_else***, ***switch_case***, ***while***, ***do_while*** and ***for-loop***, and we next consider each case

[8] Other assignment-type operators include +=, -=, *=, /=, &=, |=, ~=, %=, >>=, and <<=.

in more detail. After this control-statement swapping, there are no inlineable function callsites within a conditional-control or loop-control expression.

• ***if_then_else*** condition

Swapping the ***if_then_else*** construct is shown in Fig. 4.2. Whenever a function callsite is detected to be inside the ***if_then_else***'s condition expression, the expression is swapped out and evaluated separately. The result is written into an integer temporary variable, which then takes the place of the original ***if_then_else*** condition. A specific example of swapping the ***if_then_else***'s condition is shown in Fig. 4.3.

Note that there is an integer variable named "temp01" in the swapped program. Actually, it is a kind of "fresh" temporary variable whose name is controlled by a global counter. Whenever a temporary variable name is required and generated, the counter increases, thus guaranteeing that no duplicate names appear. This idea is frequently used through the entire thesis.

```
if(condition)
  then_statement
else
  else_statement
```

Swap ***if_then_else*** condition →

```
{ int temp01;
temp01 = condition;
if(temp01)
  then_statement;
else
  else_statement; }
```

FIG. 4.2 GENERAL RULE FOR SWAPPING AN *IF_THEN_ELSE* CONDITION

```
if( t == f(x))
    { b = 10; a++}
else
  {
  t += 10;
  r = f(x) – 5;
  }
```

Swap ***if_then_else*** condition →

```
{ int temp01;
  temp01   =   (t   ==
f(x));
  if(temp01)
      { b = 10; a++;}
  else {
   t += 10; r = f(x) – 5;
        }
```

FIG. 4.3 EXAMPLE FOR SWAPPING ***IF_THEN_ELSE*** CONDITION

```
switch(condition)
   labeled_statement;
…
```

Swap ***switch_case*** condition →

```
{
  int temp01;
  temp01 = condition;
  switch(temp01)
      labeled_statement;
}
…
```

FIG. 4.4 GENERAL RULE FOR SWAPPING ***SWITCH_CASE*** CONDITION

•***switch_case*** expression

Swapping the ***switch_case*** construct is shown in Fig. 4.4, whenever a function callsite is found within the expression part of the construct. Swapping for the construct is done in much the same way as for the *if_then_else* construct.

• ***while*** condition

While-condition swapping is performed when there is a function callsite in the condition part. C semantics require that a while's body keeps on looping whenever the ***while-condition*** is true at the beginning of each iteration. This implies that the **while-condition** needs to be evaluated before every iteration. This in turn implies that the new assignment statement cannot be outside of the loop after swapping, thus this evaluation is earlier than the evaluation of ***loop-body***. By moving the **while-condition** into the loop body and using a constant value "1" instead, the standard ***while-loop*** is converted to an unconditional ***while-loop***, with an ***if_then_else*** construct in the body, as shown in Fig. 4.5. Then the ***if_then_else*** construct is standardized as introduced before. This guarantees that the original while-condition is evaluated before every iteration and the entire loop terminates when the condition becomes false. A specific example for ***while-loop*** swapping is presented in Fig. 4.6.

```
while (condition)
  loopbody;
```

```
while (1){
  if(condition)
   loopbody;
 else
   break;
}
```

```
while(1){
  temp01 = condition;
  if(temp01)
      loopbody;
  else
      break;  }
```

FIG. 4.5 GENERAL RULE FOR ***WHILE-LOOP*** SWAPPING

```
while(f(i) < 10){
  i++;
  f(i);
}
```

```
while(1){
  if(f(i) < 10) {
   i++;
   f(i);
  }
else
   break;  }
```

```
while(1){
   int temp01;
   temp01 = f(i);
    if(temp01){
       i++;
        f(i);  }
    else break; }
```

FIG. 4.6 EXAMPLE FOR SWAPPING ***WHILE-LOOP***

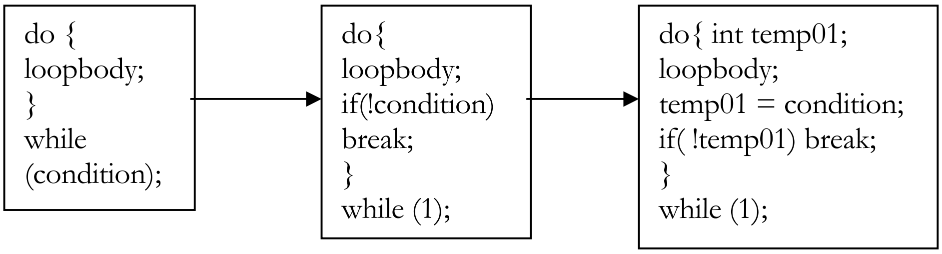

FIG. 4.7 GENERAL RULE FOR SWAPPING ***DO_WHILE*** CONSTRUCT

• ***do_while*** condition

The rule for swapping the condition from a ***do_while*** loop is illustrated in Fig. 4.7. It is very similar to the rule for swapping the ***while-loop***'s condition.

• ***for-loop*** condition

A ***for-loop*** will be swapped if a callsite is detected in any of the three looping-control parts. The ***for-loop***'s swapping is shown in Fig. 4.8 and Fig. 4.9. As shown, ***for-loop*** swapping is complicated. First, the ***for-loop*** is converted into a standard ***while-loop*** with the initializer swapped out of the loop and the modifier moved into the loop. Second, the standard ***while-loop*** is processed as before.

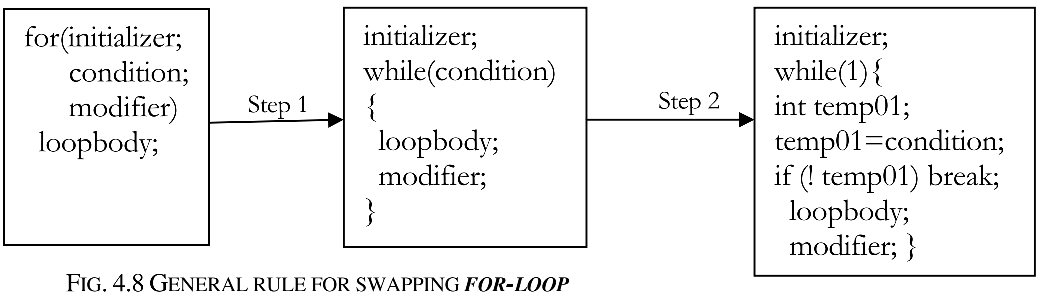

FIG. 4.8 GENERAL RULE FOR SWAPPING ***FOR-LOOP***

Note that after each ***for-loop*** swapping, the corresponding "***for***" construct has been completely replaced by an equivalent ***while*** loop construct. This significantly changes the parse tree, but does not affect semantics and should not affect efficiency.

```
for(i=0;
   i<f (10);
   i++){
printf ("%d",i);
}
```

Step1 →

```
i = 0;
while (i<f(10)){
 printf ("%d",i);
 i++;
}
```

Step2 →

```
i = 0;
while (1){
  int temp01;
  temp01=(i<f (10));
  if(!temp01) break;
  printf ("%d",i);
  i++; }
```

FIG. 4.9 EXAMPLE FOR ***FOR-LOOP*** SWAPPING

4.3.2 Swapping Return Statements

Return statements also need to be swapped if the return expression contains a function callsite. The original return statement is replaced by one assignment statement followed with a jump statement, as shown in Fig. 4.10 and Fig. 4.11. The function_type is the current function's return type, which is easily obtained from the parse tree[9].

```
…
return expression;
…
```

Swap return statement →

```
…
function_type temp01;
temp01 = expression;
return temp01;
…
```

FIG. 4.10 GENERAL RULE FOR *RETURN*-STATEMENT SWAPPING, ASSUMING THAT THE EXPRESSION CONTAINS A CALLSITE

[9] This is a case where an "upward" broadcast introduced in Chapter 3 is useful.

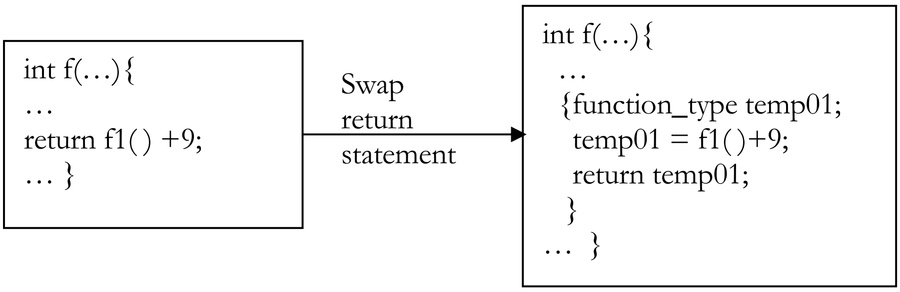

FIG. 4.11 EXAMPLE FOR *RETURN*-STATEMENT SWAPPING

4.3.3 Swapping Callsites in Declarations

C allows variables to be assigned initial values at declaration time. If callsites are contained in an initializing statement, they cannot be inlined directly and thus need to be swapped. The rule about how to swap function callsites between declarations and statements and an example are presented in Fig. 4.12 and Fig. 4.13 respectively. As shown, a simple algorithm is used. A scan is made over the entire parse tree, trying to find variables initialized at declaration time. We then retain the declaration of the variable name, but remove the initialization and put it into a FIFO queue of parse trees. When finished scanning the current declaration block, we insert the queue's contents at the very beginning of the current statement block. This guarantees the evaluation order is correct and all data dependencies are properly maintained.

Note that whenever a variable declaration has an initialization expression containing a callsite, the *entire* current declaration block must be swapped as introduced above. In case we try to swap only those variables with callsites in their initializations and ignore others, data dependencies may be violated. The example in Fig. 4.14 illustrates this.

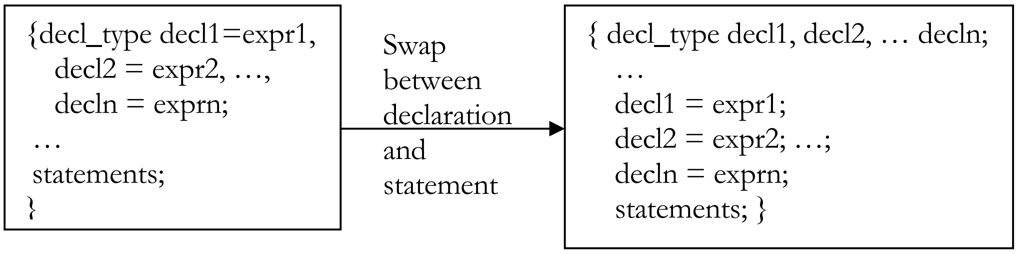

FIG. 4.12 GENERAL RULE FOR SWAPPING VARIABLE DECLARATIONS THAT HAVE INITIALIZATIONS

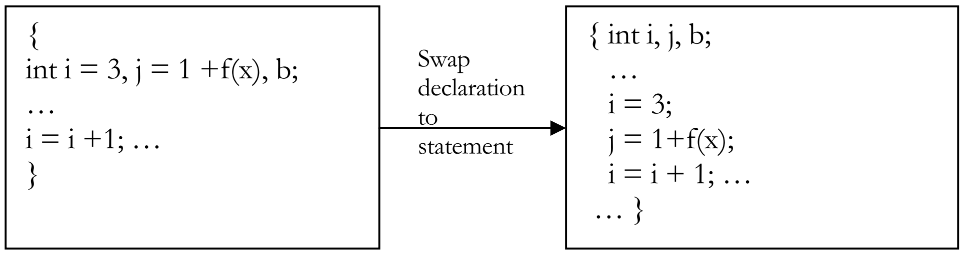

FIG. 4.13 EXAMPLE OF SWAPPING VARIABLE DECLARATIONS THAT HAVE INITIALIZATIONS

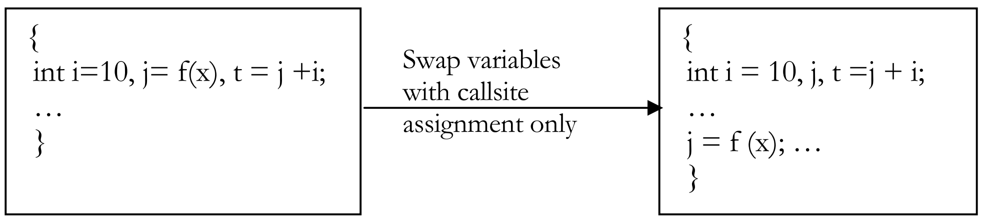

FIG. 4.14 EXAMPLE OF INCORRECT SWAPPING OF INITIALIZATIONS

As shown above, on the left, *t* is assigned the value of "*j+i*" before swapping, where *j* has the value of f(x). But if we swap only the variables with function calls in their initialization, as shown on the right, *t* is assigned the value of "*j+i*", where *j* is not initialized and thus has an unknown value. This swapping will break the data dependency and must be avoided.

This concludes the description of swapping. When all these forms of swapping finish, it is impossible to find an inlineable callsite inside any conditional expression, a loop control expression, a return statement or a declaration. We are left with the possibility that a callsite might be nested inside an expression, and how this is handled is discussed next.

4.4 Splitting

Splitting is used to rewrite an expression containing inlineable callsites into a series of standard form statements, where all callsites appear to be in the standard formats. It is difficult to do this properly. There are six types of splitting to consider.

4.4.1 Removing Comma Operators

It is valid C to join a series of expressions with comma operators. Whenever a function callsite is detected to be in one of the comma expressions, the entire comma expression must be split, as presented in Fig. 4.15.

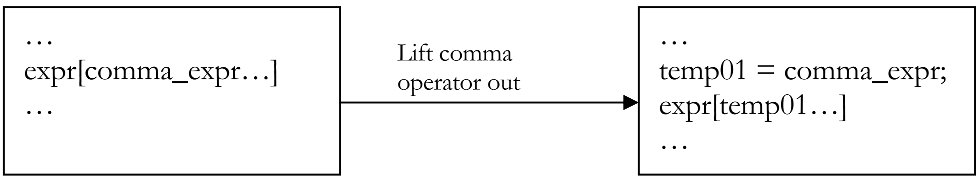

Fig 4.15(a) Rule for lifting comma operator

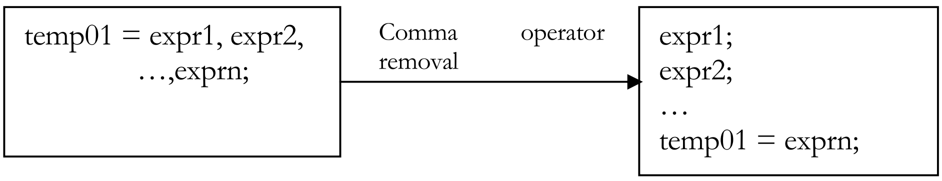

Fig 4.15(b) Rule for removing comma operator

```
…
t = (f1(2), r = f(f(s)), l++);
…
```

Comma operator removal →

```
…
f1(2);
r = f(f(s));
t = (l++);
…
```

Fig 4.15(c) Example for removing specific comma operator

FIG. 4.15 COMMA-OPERATOR REMOVAL

The expressions are separated with commas and it is the inliner's responsibility to replace them with semicolons. First, the comma operators nested in an expression are lifted out to the statement level. Then, the comma operators will be standardized. There are several places of actions in the parser where the parse-nodes will be marked inside comma-operators when they were created. Note that this technique could only be applied within marked *comma operator* expressions where comma operators appear in a place as any other binary operators. The commas separating variable declarations or function callsite parameter lists are not affected.

4.4.2 Splitting Short-cut Operators

Special approaches are needed when splitting two binary operators – logical-and ("&&") and logical-or ("||") and these approaches were introduced by [5]. Compared with all other binary operators, they have different properties and need to be considered separately.

- Splitting logical-and ("&&") operator

As with comma operators, the shortcut operators also need to be lifted first, in case they are deeply nested in an expression. The expression containing logical-and ("&&") operators will then be split into a kind of nested ***if_then_else*** statement, as shown in Fig. 4.16(a), Fig. 4.16(b) and Fig. 4.16(c).

```
…
expr(… (expr1 && expr2) …)
…
```

Lift '&&' operator →

```
…
{
int temp01;
temp01 = (expr1 && expr2);
expr(… temp01 …)
}
…
```

Fig 4.16(a) Rule for lifting logical-and operator

```
varname =
expr1 && expr2;
```

Split logical-and shortcut operator →

```
{ int temp01=0;
  int temp02;
  temp01 = expr1;
  if (temp01){
    temp02 = expr2;
  if (temp02)
    temp01 = 1;  }
  varname = temp01;
}
```

Fig 4.16(b) Rule for splitting logical-and operator

```
t = (f1(x) && (f2(b)+10));
```

Split logical-and shortcut operator →

```
{ int temp01 = 0;
  int temp02 = f1(x);
  if(temp02){
   int temp03 = f2(b) + 10;
    if(temp03)
       temp01 = 1; }
   t = temp01;   }
```

Fig 4.16(c) Example for splitting logical-and operator

FIG. 4.16 EXAMPLE FOR SPLITTING LOGICAL-AND OPERATOR

ANSI C requires that shortcut evaluation be performed on logical-and operations strictly from left to right. If the left operand evaluates to "false", the result is false and it is unnecessary to evaluate the right operand. The logical-and splitting technique properly mimics this evaluation.

• Splitting logical-or ("||") operator

The rule for lifting logical-or ("||") operator is exactly the same as that used in lifting logical-and ("&&") operator. The rule used to split logical-or operator is similar to that of logical-and splitting and is shown in Fig. 4.17.

```
varname =
expr1 || expr2;
```

Splitting logical-or shortcut operator →

```
{ int temp01 =0;
  int temp02;
  temp01 = expr1;
   if( !temp01){
      temp02 = expr2;
      if(temp02) temp01 = 1;}
   varname = temp01;      }
```

FIG. 4.17 RULE FOR SPLITTING LOGICAL-OR OPERATOR

Note that we only show the technique to split one shortcut operator here. In case shortcut operators are nested, the same technique will be applied again, until all shortcut operators are removed. Also note that if the final assignment expression does not exist (in Fig. 4.16, if ***t*** does not exist), the actual type the shortcut evaluation returns can be decided by applying the ***typeof*** operator, a GNU [14,24] extension, on the entire evaluation expression. Since this extension is not portable (other C compilers do not support this feature) and this situation is expected to be unusual, plus there are no other situations where the inliner needs to perform type inference, the inliner actually does not process such callsites, marking them uninlineable.

4.4.3 Splitting (condition) ? (expression) : (expression)

The special conditional-evaluation construct (?:) could be converted into an ***if_then_else*** statement, if it is not nested in any other expression, as shown in Fig. 4.18. However, compared to all the other kinds of splitting, the practical examples are a bit complicated, as presented in Fig. 4.19. Whenever the special conditional evaluation is detected and it is found that there are function callsites within any of its structures, the entire construct will be split first, as shown in Step 1. This separates the conditional construct from other nested structures, for instance, from within a nested expression. Step 2 shows the way to rewrite the structure into the standard ***if_then_else*** construct. Note that the ***if_then_else*** condition might also be swapped out at this stage, as introduced in ***if_then_else*** condition swapping section.

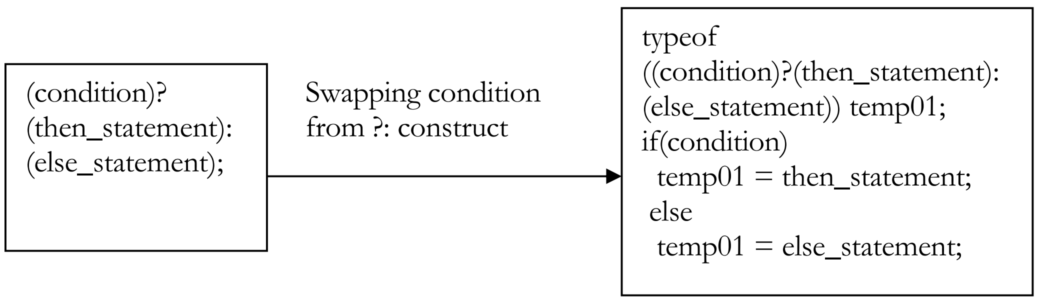

FIG. 4.18 GENERAL RULE FOR SPLITTING ?: OPERATOR

```
t =
((f(x) == 1)?
(f(x)) : (x++)) –10;
```

Original form

```
{ type_of_t temp01;
  temp01 = (f(x) ==1)?(f(b)):(x++);
  t = temp01 – 10;
 }
```

Step 1

```
{ type_of_t temp01;
  { int temp02;
    temp02 = (f(x) ==1);
    if(temp02)
      temp01 = (f(b));
     else
      temp01 = (x++);
   }
  t = temp01 – 10;   }
```

Step 2

FIG. 4.19 EXAMPLE FOR SWAPPING ?: OPERATOR

4.4.3 Splitting Expression

All function callsites within an expression (except those nested in short-cut operators) should be split out if the expression is not in the standard format yet. This is presented in Fig. 4.20 and Fig. 4.21.

As shown in Fig. 4.20, every callsite within the expression needs to be swapped out and evaluated before the expression's evaluation. As well, a corresponding variable is written into the callsite's place.

```
…
expr(…callsite_1…
        callsite_n);
…
```

Split
expression

```
…
{ typeof_callsite_1 temp01;
   …
   typeof_callsite_n temp0n;
   temp01 = callsite_1; …
   temp0n = callsite_n;
   expr(…temp01…temp0n); }…
```

FIG. 4.20 GENERAL RULE FOR SPLITTING EXPRESSION

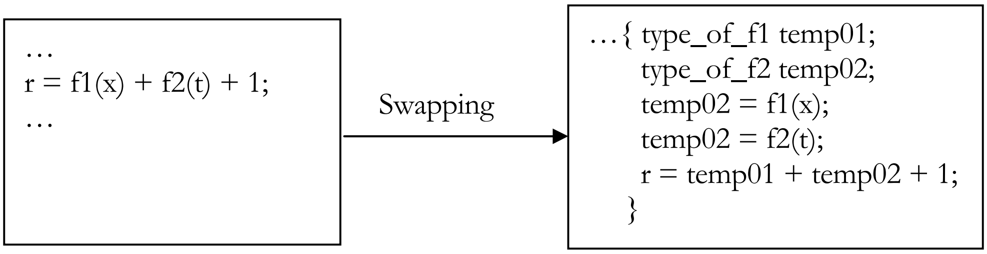

FIG. 4.21 EXAMPLE FOR SPLITTING EXPRESSION, ASSUMING THERE ARE CALLSITES IN THE EXPRESSION

Also shown that the original expression is converted into a series of subexpressions where function callsites are split out and evaluated separately. Note that the temporary variable's return type is determined by the function being invoked and the necessary information is obtained easily by analyzing the parse tree. However, there might be some unexpected side effects during this splitting, details will be discussed later.

When this process finishes, it is guaranteed that there is no function callsite within any kind of expression (except nested callsites). However, there still could be nested function callsites after this splitting.

4.4.4 Split Nested Callsite

All callsites nested within other callsites (except for those nested in short-cut operators) must be split in order to produce the standardized formats. Examples are given in Fig. 4.22 and Fig. 4.23. When a C compiler generates instructions to evaluate callsites, the processing order is leftmost innermost[10]. Note that this evaluation order is not part of the ANSI C standard, but all compilers we tested used the same order to evaluate

---

[10] Pick up a function callsite from left to right, with the deepest nested calls first.

expressions and callsites' parameters. It is desirable to keep the same order when performing swapping and splitting, so that the standardized format could still produce the same result as the original program if it (erroneously) relies on the left-to-right evaluation order. After this splitting, all nested callsites have been removed and thus standardized formats created.

```
callsite1(callsite2(calsite3(.. )));
/* all callsites are inlineable */
```

Splitting nested callsites →

```
temp01 = callsite3(..);
temp02 = callsite2 (temp01);
callsite1 (temp02);
/* all callsites are in standard formats now */
```

FIG. 4.22 GENERAL RULE FOR SPLITTING NESTED CALLSITES

```
s = f3 (f2(f1(10)));
```

Specific splitting for nested callsite →

```
{ type_of_f1 temp01;
  type_of_f2 temp02;
  type_of_f3 temp03;
  temp01 = f1(10);
  temp02 = f2(temp01);
  temp03 = f3(temp02);
  s = temp03;
}
```

FIG. 4.23 EXAMPLE FOR SPLITTING NESTED CALLSITES

This splitting specifically processes the nested callsites, no matter where they are swapped or split from. Nested function callsites are broken and rewritten into a series of standard format callsites and dependent relationships are carefully maintained.

4.4.5 Splitting Deeply Nested Callsites

It has been difficult to decide whether we should swap and split callsites that are deeply nested inside an expression, because both the evaluation order and the implicit effect on other callsites might actually make the swapping/splitting fail, as presented in Fig. 4.24.

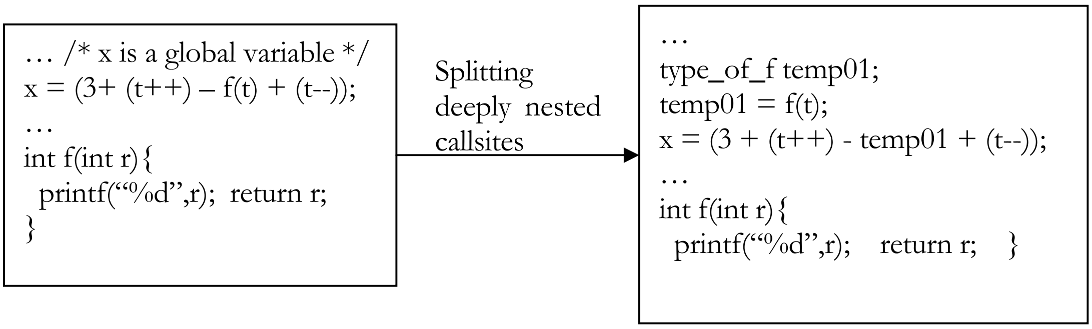

FIG. 4.24 EXAMPLE FOR INCORRECTLY SPLITTING DEEPLY NESTED CALLSITE IN EXPRESSION

As shown in Fig. 4.24, the evaluation might produce incorrect results after splitting, because the original expression erroneously relies on the evaluation order (actually, relies on a specific compiler). Since C standard does not specify the evaluation order[11] for binary operators, it is a compiler implementation's policy to decide the order of evaluation and any presumed order of evaluation should be avoided. In this case, we make a simple check on the expression. This checks whether any callsite argument appears more than once in the original expression. This also checks whether there are multiple callsites in the expression. If any of the checks work, we would assume that the

---

[11] Except for logical-and ("&&") and logical-or ("||"), whose order is specified strictly from left to right.

callsite arguments might have been implicitly modified and any callsite inside such an expression should be avoided by swapping/splitting.

4.4.6 Swapping/Splitting Order

This ends the description of the swapping and splitting process. After splitting, all callsites appear to be in the standard format as desired and are ready to be inlined. During the swapping and splitting process, the parse tree would be frequently locally reparsed as discussed in Section 3.7.2. However, the order of swapping and splitting is especially important and needs to be specified separately.

When applying these rules, we assume that all swapping is done first. Each swapping process is only invoked once, and it scans the parse-tree and strictly follows the order as introduced in Section 4.3. However, in some cases, the result of splitting is dependent on the order of splitting. An example is given in Fig. 4.25. Thus the splitting processes the parse tree using a leftmost-outermost order. Every possible structure in the parse tree is explicitly checked to see if one of the splitting rules applies. Whenever a match is found, the corresponding technique will be applied and the parse tree will be rewritten. This process will not stop until each callsite has been rewritten into its corresponding standard format.

As shown in Fig. 4.25, the callsite *printf* is split by "one-level callsite parameter splitting" first, because it is the leftmost-outermost available splitting. However, if we use different processing order, the final result is not easy to predict. Our splitting order generates a series of expressions that contain other kinds of swapping/splitting constructs, such as short-cut operator, nested callsites, as shown in Fig. 4.25(b). Thus the swapping/splitting process will continue (shown in Fig. 4.25(c) for subtree ∂ which is a short-cut operation splitting and Fig. 4.25(d) for subtree • which is another one-level callsite splitting). Note that if the function's return type is not available[12](e.g. type_of_printf), it is assumed to be integer as default. Each rule will simplify the original or generated structure that it applies on, and processing will not stop until every callsite comes to be in its standard format.

```
printf("…",
  f1(f2(…)&& b),
  f2(printf("…")));
```

Order of splitting →

```
{type_of_f1 temp01;
 type_of_f2 temp02;
 temp01 = (f2(…) && b); ∂
 temp02=
   f2(printf("…"));          •
printf("…",temp01,temp02);
}
```

Fig 4.25(a) program before callsite standardization

Fig 4.25(b) callsite standardization for source code.

```
…
{ int temp03 =0;
  type_of_f2 temp04;
  temp04 = f2(…);
  if(temp04) {
   if(b)) temp03 = 1; }
  temp01 = temp3;
} …
```

```
…
{ type_of_printf temp05;
  temp05 = printf("…");
  f2(temp05);
 }
…
```

[12] In… ted and recommended. Howe… nction without its prototype declaration.

Fig 4.25 (c) callsite standardization for subtree ∂

Fig 4.25(d) callsite standardization for subtree ●

FIG. 4.25 ORDER OF SWAPPING AND SPLITTING PROCESS

4.5 Inlining a Callsite

Recall that inlining requires that a function callsite be replaced by a copy of the subprogram body, with parameter-passing simulation, return statement processing, identifier-conflict resolution, callsite removal etc. as introduced in Chapter 2. We assume that each callsite is in one of the standard formats and examine each of the inlining requirements in detail.

1. Body Duplication

The selected callsite's function body is duplicated by explicitly cloning the function's definition as recorded by the Function List. Details on function cloning and the message system can be found in Chapter 3.

2. Parameter-Passing Simulation

Parameters are passed to the duplicated function body by creating local variables in the cloned body and assigning actual callsite arguments to them. Note that the simulated parameters are placed ahead of the local variables in the original function body.

```
int i = 4;
void f(int i, int t){
/* f's body */
}
…
f( 3,i );   /* callsite */
…
```

Parameter passing

```
…
{
int i = 3;
int t = i; /* wrong i */
/* duplicated body */
 }
…
```

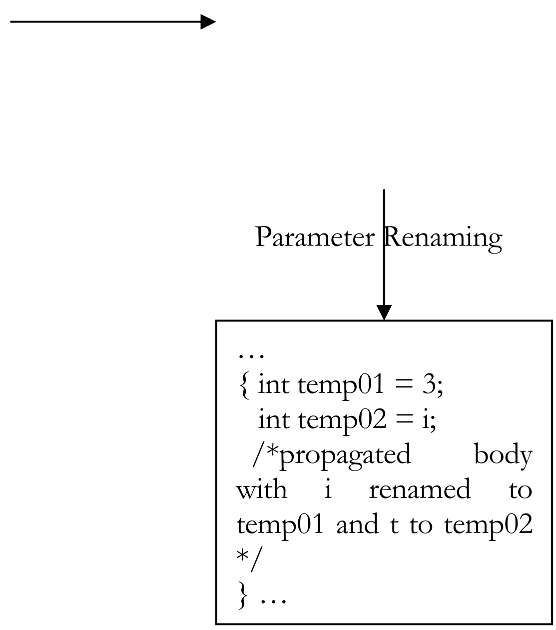

FIG. 4.26 EXAMPLE FOR PARAMETER NAME CONFLICTS AND RENAMING

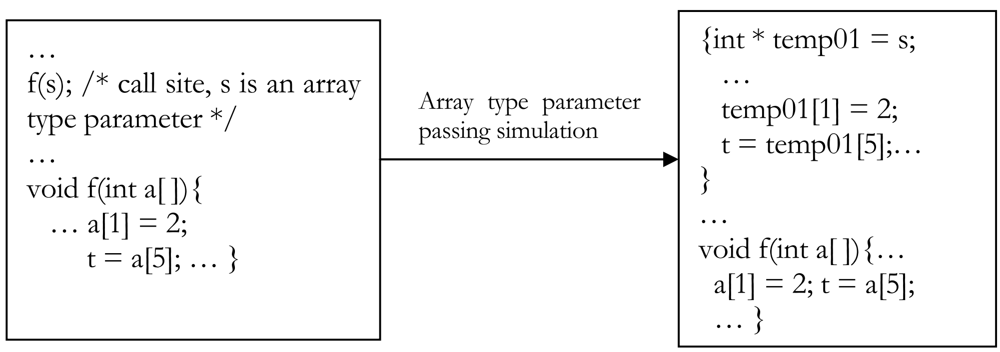

FIG. 4.27 EXAMPLE FOR ONE-DIMENSIONAL ARRAY PASSING AND RENAMING

3. Renaming

There is a possibility that one of the callee's parameters has the same name as a variable in the actual parameter used in the callsite or the callee's local variable. An example is given in Fig. 4.26. Rather than detecting such a conflict and renaming it as in [5], it is easier and harmless to blindly perform formal-parameter renaming. This involves creating a distinct local-variable name, assigning the local variable with this new name and propagating the new name throughout the duplicated function body. However, some parameters have special properties and need more processing than just renaming. Special efforts are thus made when passing array-type parameters.

- One-dimensional array

  When function's formal parameter list is detected to have a one-dimensional array in its parameter list, the inliner will use a pointer with the same element type and initialize it with the address of the array parameter. A local renaming and replacement follow this, as shown in Fig. 4.27.

- Multi-dimensional array

  A multi-dimensional array is simulated by a pointer to an array that has one less dimension. A case and its solution are illustrated in Fig. 4.28. This mechanism uses a "***typedef***" operator to create an array type that has *exactly* one less dimension. Then, it will use this type to declare a pointer variable and initialize it with the name of the array being passed. All the multi-dimensional array accesses within the function body are thus redirected to this pointer, as with the single-dimensional array.

4. Return Simulation

To mark the exit point for the duplicated function body, a distinct label is always created and inserted at the very end of the duplicated body as described in Chapter 2. All the return statements in the duplicated function body have to be replaced by an assignment statement followed by a ***goto*** statement, as shown in Fig. 4.29.

```
…
 f(b);    /* callsite , b is
int[10][50] type */
…
void f(int a[10][50]){/*
function */ …
  a[1][1] = 1;
  t = a[10][5];
 …}
```

Multi-dimensional array parameter passing simulation →

```
{typedef int temp01[50];
  temp01 * temp02 = b;
  …
  temp02[1][1] = 1;
  t = temp02[10][5];
   … } …
void f(int a[10][50]){/*
function */ …
  a[1][1] = 1;
  t = a[10][5];
 …}
```

FIG. 4.28 MULTI-DIMENSIONAL ARRAY PASSING SIMULATION

```
int f(int t){ /* function */
   int i  = 5;
   …
   return i + t;
}
…
RT_002 = f(25); /* callsite */
…
```

Return statement simulation →

```
{  /* inlined body */
 int temp01 = 25; // param
 int RT_003; // return value
 int i = 5;   // local variable
 …
 { RT_003 = i + temp01;
   goto exit_01;
 }
 exit_01: RT_002 = RT_003;
}
```

FIG. 4.29 EXAMPLE FOR RETURN-STATEMENT PROCESSING

5. Label Renaming

If there are labels in the function body and the function is directly recursive, there will be label-name conflicts after the recursive callsite is inlined, because C only allows distinct labels. An example is given in Fig. 4.30. A scan is performed on the cloned function body to do a blind local replacement of all labels. This harmlessly removes the possibility of multiply defined labels.

6. Specialization Opportunity

If there is a constant value in the callsite parameter list and the corresponding formal argument happens not to be modified within the function body, there is an opportunity for the corresponding formal parameter to be replaced by the constant value. This is an example of a specialization opportunity, and it enables some other compiler optimizations. For instance, it enables constant merging: when an optimizing compiler finds that two constant operands are available for a binary operator, it performs a compile-time evaluation and replaces the original expression with the evaluated constant value.

```
int f(int i){
 int result;
 if(i == 0){
      result = 0;
      goto outer;
   }
 else
    result = f(i-1);
 outer: return result;
}
…
f (10); // inline here
…
```

Fig 4.30(a) Function definition and callsite

```
{ int i = 10;
  int result;
  int RT_0001;
  if( i == 0){
             result = 0;
             goto outer;
             }
  else
   result = f(10-1);
  outer: { RT_0001 = result;
            goto exit_01;
         }
 exit_01:  }
```

Fig 4.30(b) One inlining step without label renaming

```
{ int i = 10;
  int result;
  int RT_0001;
  if( i == 0){ result = 0;
              goto exit_02;
              }
  else  result = f(10-1);
  exit_02: {
    RT_0001=result;
    goto exit_01; }
 exit_01:
…}
```

Fig 4.30(c) One step inlining with label renaming

FIG. 4.30 EXAMPLE FOR LABEL RENAMING

When a constant value is found in the callsite parameter list, a simple analysis is applied on the cloned function body, involving two steps:

1. Check to see whether the formal parameter is modified directly. This checks whether the parameter name appears on the left-hand side of an assignment operator or has an increment/decrement operator applied.
2. Check to see whether the parameter address is fetched. This checks whether the parameter appears immediately after the operator "&".

If both checks fail, the inliner will assume that the constant variable is not modified in the function body, and a local constant propagation will exploit the specialization opportunity, as presented in Fig. 4.31. In this example, the analysis also finds that local variable ***i1*** 's address is fetched and local variable ***i2***"s value is modified directly, and thus both are removed from the constant-propagation candidate list and only variable ***i3*** is constant propagated. The current implementation does not finish the removal of the constant-propagated local variables, although it could be done easily.

Note that a pointer might just fetch the variable's address but never modify its value. This still makes the inliner believe that the variable cannot be constant propagated. The effect is slightly harmful to further optimizations, but does not affect inlining or semantic correctness, and thus can be ignored.

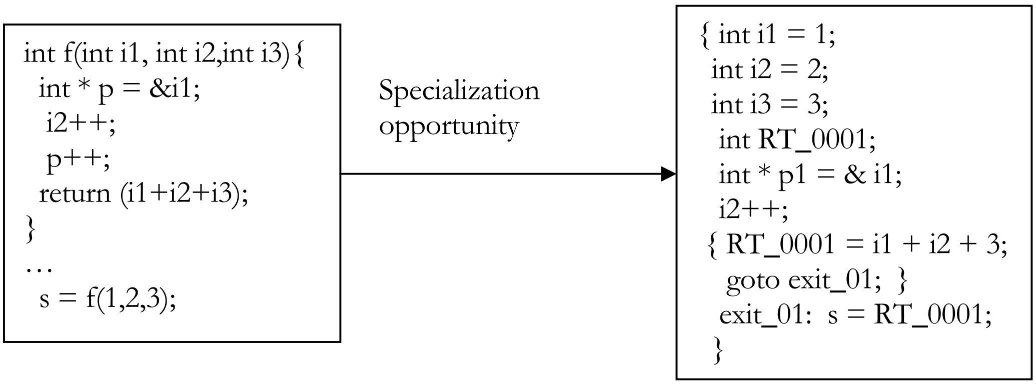

FIG. 4.31 EXAMPLE FOR SPECIALIZATION OPPORTUNITY FOR VARIABLE *I3*

7. Callsite Removal

After the cloned function body has been properly processed, it will be used to replace the callsite. At the same time, the callsite is marked "inlined" and all further output is redirected to the substituted function body. A detailed description can be found where the proxy pattern design discussed in Chapter 3.

This ends the discussion of inlining techniques, which have been fully implemented in our inliner. The problem of how to decide which callsite should be inlined still remains. In the next chapter, we give a detailed discussion of the inlining decision algorithm.

Chapter 5

Inlining Decision Algorithm

This chapter covers the inlining-decision algorithm, which will be applied on the standardized parse tree. First, removal of uninlineable callsites is discussed. Second, special call-graph generation and profile-information collection is discussed. Then, a multi-technique inlining-decision algorithm is introduced, along with discussions about code-size approximation and version issues. Considerations are also given to cache and recursion issues. Finally, experimental results are presented to illustrate the effectiveness of the algorithm.

5.1 Uninlineable Callsite Elimination

The inlining technique introduced in Chapter 4 has an obvious and necessary limitation: the function's definition *must* be available at the time of inlining. Initially, all callsites are placed into a list of possible candidate callsites for inlining. In many programs, there are callsites whose callees' definitions are not available or where the actual callee cannot be determined at compile time, and those callsites cannot be inlined. These callsites include ones invoking library routines and ones using function pointers. In such cases, the callsite must be removed from the list of possible inlining candidates.

5.1.1 Removal of Library Routines

The entire parse tree is scanned once to remove from the list all the callsites whose source code is not available – routines not explicitly defined within the source code. This is achieved by scanning the Function List first, collecting all the function names whose definitions are available. Then, for every callsite in the current parse tree, its callee's name is compared with the function names collected from the Function List. In case it is not found, the callsite will be marked uninlineable and permanently removed from the list of inlining candidates.

5.1.2 Removal of Function Pointers

Callsites using function pointers cannot be inlined because the function really invoked by the callsite at run time cannot always be determined at compile time, as shown in Fig. 5.1. In this figure, we cannot tell whether *f1()* or *f2()* will actually execute at run time, even if both the functions' definitions are available. Thus callsite of *p->r()* cannot be inlined.

```
…
if(i<10) p->r = & f1;
 else     p->r = &f2;
 p->r(10);
…
void f1(int a){/* body of f1 */ }
void f2(int t){/* body of f2 */ }
```

FIG. 5.1 EXAMPLE OF FUNCTION POINTER

The function-pointer elimination consists of one scan: function callsites appearing on the right hand side of a dereferencing operator (* and ->) will be permanently removed from the list of inlining candidates.

5.2 Call-Graph Generation

The program call graph is used to represent the functions and their callsites. We do not use the parse tree because the inlining-decision algorithm is only concerned with callsites, while the parse tree has much more detailed information on program constructs that are not directly related to the decision algorithm. A formal definition of call graph is given below.

5.2.1 Program Size and Call Graph

The call graph for a program $\mathbf{P}$ is a labeled, directed multi-graph, with a node for each function $\mathbf{P}_i$. Every arc a = $(\mathbf{P}_i, \mathbf{P}_j)$ within the graph indicates an occurrence of a function callsite invoking $\mathbf{P}_j$ (callee) in function $\mathbf{P}_i$ (caller). A SYSTEM (operating system and run-time environment, modeled by another node) node is also introduced, which simulates the behavior of the invocation relationship between the operating system and user program. Actually, we have less interest in the callsites that could be invoked by SYSTEM, because SYSTEM could implicitly invoke any function and these invocations cannot be inlined (for example, in any case, function “main” is likely to be invoked once and should not be inlined).

Consider inlining an arc a = ($\mathbf{P}_i, \mathbf{P}_j$): a new version of P_i is created, say $\mathbf{P'}_i$. The size of $\mathbf{P'}_i$ is generally less than size ($\mathbf{P}_i$) + size ($\mathbf{P}_j$), due to the elimination of space overhead in parameter passing and increased opportunities for optimization. Moreover, if $\mathbf{P}_j$ is a leaf node and there is no other invocations for $\mathbf{P}_j$ in the program after inlining, $\mathbf{P}_j$ could be eliminated as dead code. Hence, sometimes the code after inlining can, in reality, be smaller than it was before. However, for simplicity, we pessimistically model the new code size of the caller as the sum of the caller's size and the callee's size, and we assume that no function is omitted after inlining.

5.2.2 Call-Graph Generation

From the call-graph introduction, we know that all uninlineable callsites will presumably not be considered for inlining, and thus they do not need to appear in the call graph. Our form of call-graph is a two-dimensional dynamic array of Tree pointers. The first dimension is for the functions whose definitions appear in the source program (including the *main* function). The second dimension distinguishes between callsites in each function. The array entries are actually shared pointers to the callsites in the parse tree and the callsites appear in the dynamic array according to their textual order in the original standardized program. A simple program and its call graph are given in Fig. 5.2. As shown in the figure, there are three function definitions – *main*, *f* and *f1*. So, there are three entries in the first-level call graph, shown on the right. Similarly, there are two invocations in function *main* – *f* and *f1*, and the call graph properly represents this feature. Note that every node in the call graph is a shared Tree pointer whose contents are kept in the parse-tree. Details could be found in Section 3.5.

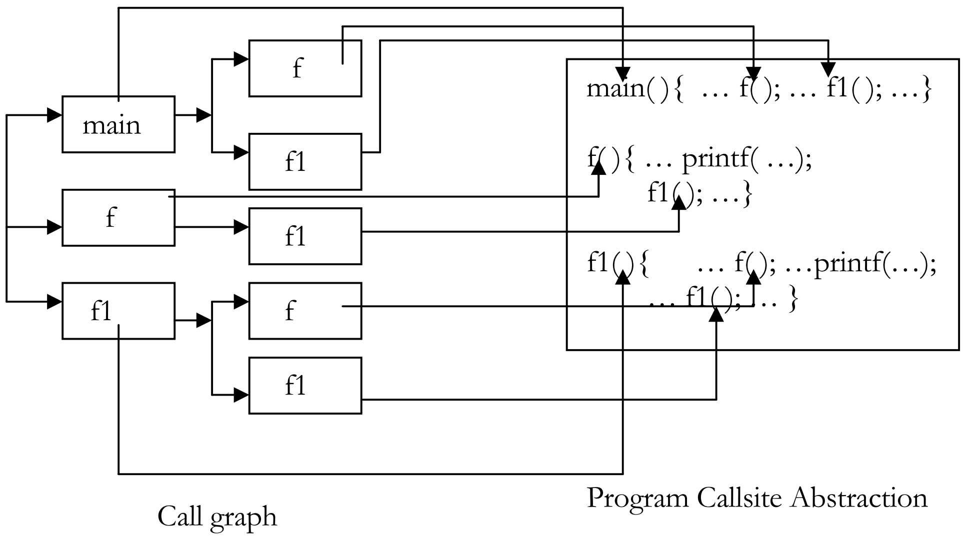

FIG. 5.2 EXAMPLE OF A STATIC COMPRESSED CALL GRAPH

This call-graph design shares callsites between the parse tree and the call graph, and thus smoothly connects the call graph with the parse tree. When applying the decision algorithm, it is easy to obtain information directly related to a specific callsite from the parse tree. Therefore, we will use only the call graph for the description of inlining-decision algorithm.

5.3 Profile-Information Collection

In order to get the accurate callsite profile information that the decision algorithm needs, the current standardized parse tree is duplicated, special dummy functions are inserted, and a source program is restored from the parse tree. Then that program is compiled, executed and profiled. The profiler tool, gprof [14], has a built-in limitation: it does not distinguish between two or more different callsites in $\mathbf{P}_i$ where $\mathbf{P}_j$ is called. Fig. 5.3 shows this situation. As illustrated in the figure, there are two *f1( )* callsites invoking same function *f( )* and they are invoked 15 times together. However, the profiler cannot let us determine how many times each individual callsite is invoked.

The solution is to insert distinct dummy functions. A dummy callsite is inserted to immediately follow each callsite in the standardized source code. An example is given in Fig. 5.4.

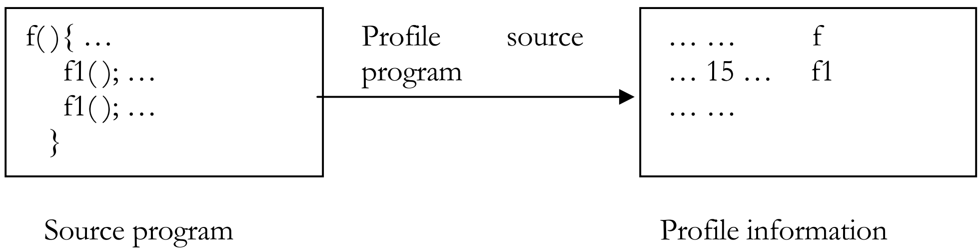

FIG. 5.3 BUILT-IN LIMITATION OF PROFILER

Each dummy function has a distinct name, a void return type and an empty function body. As shown in the figure, each function callsite is replaced by a block that contains the original callsite and a distinct dummy function. By analyzing the invocation frequency of a callsite's corresponding dummy function, accurate callsite-execution frequency is easily obtained from the profile and written into the call graph. To enable this profiling approach, dummy-function prototypes and definition bodies are inserted into the Function-Prototype List and Function List respectively. It is relatively easy to obtain the number of times each function was entered by analyzing the flat part of the gprof profile data.

Note that the profiled program is expected to run more slowly than the original program. When the source code is compiled with profiling enabled[13], the compiler will insert instructions to the executable, marking and timing each invocation [14]. Moreover, we have inserted calls of dummy functions to accurately profile callsites and these calls consume run-time resources. However, in this inlining system, the program is being run internally for profile collection before inlining. Profiling overhead will not occur in the execution of the final inlined program.

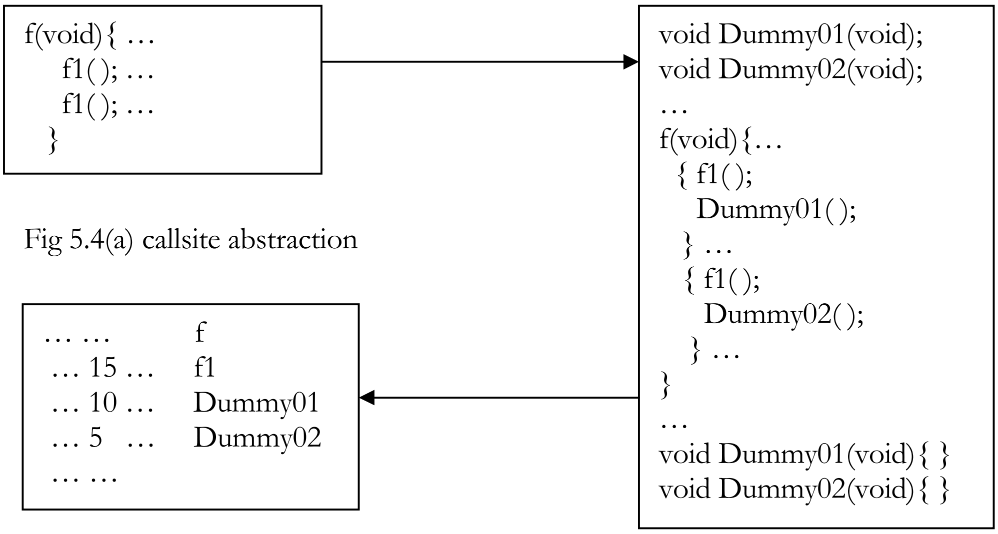

Fig 5.4(a) callsite abstraction

Fig 5.4(c) Dummy function profiling

Fig 5.4(b) Dummy function insertion

FIG. 5.4 ACCURATE CALLSITE FREQUENCY PROFILE ANALYSIS

[13] -pg for gcc command line compiler. Borland, Microsoft and Watcom all have integrated development environment, where the user needs to check the "profile info into obj" button in compiler options.

5.4 Inlining Decision Algorithm

The basic idea guiding our inlining decisions is to follow a greedy call-reduction approach, modified for cache effects and constant propagation. Since prediction of code-size expansion requires the information about the sizes of the routines and since the algorithm is partially based on version issues, function-size approximation and the version issue will be introduced first. The inlining-decision algorithm is then described, followed with an explanation of the algorithm and detailed discussion on modeling cache performance and dealing with recursive functions.

5.4.1 Function-Size Approximation

It is necessary to know the actual size of each function because this information is necessary for the inlining algorithm to make decisions. The original plan was to analyze the compiler-generated object files, subtracting the function's entry address from the function's ending address. This would lead to accurate function sizes, in byte. However, this approach was eventually rejected for the following reasons.

First, analyzing the object code format requires much more effort than expected. Although Intel Corporation [15,16] published both .obj object-file format and Unix executable language format (ELF) specifications, it is still not easy to obtain accurate routine sizes because of the formats' complexity. Second, it makes the system less portable. To analyze object files explicitly and get the exact routine size for different platforms, machine-dependent code would need to be written. This diminishes the portability that the project desires.

The solution is code-size approximation. Every terminal node is assigned size "1", and each non-terminal node's size is the sum of all its subtrees' sizes, plus a preassigned weight. Two examples are presented in Fig. 5.5 and Fig. 5.6. The complete table for parse-node size approximation is given in Appendix 3.

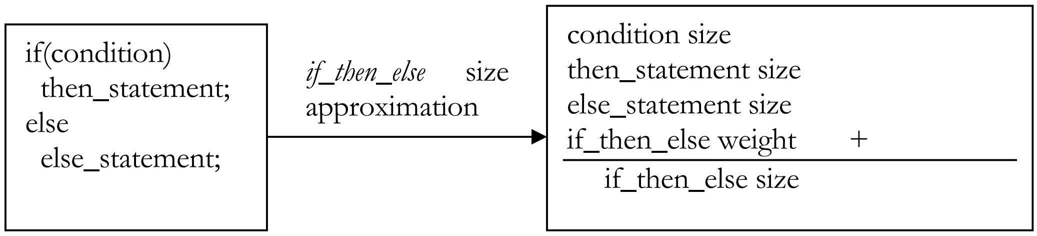

Fig 5.5 (a) General rule for size approximation of the *if_then_else* construct

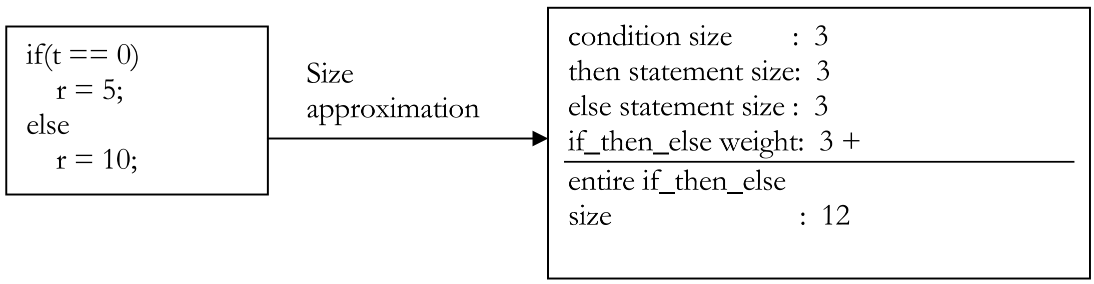

Fig 5.5 (b) Example of size approximation for an *if_then_else* statement

FIG. 5.5 PARSE-NODE SIZE APPROXIMATION

As shown in the figure above, in the *if* condition, each part of the BinaryNode (*t == 0*) is a terminal node of size 1. So the *if* condition's size is 3. The *then* statement and *else* statement are similar. The *if_then_else* construct always has a default weight of 3, so the entire construct's size is 12.

Fig. 3.6 presents the size approximation for a simple function. The function definition's size is the sum of all its subtrees: return type, function name, parameter list and function body, plus a function definition weight of 5. All separate parts' sizes and the evaluated function size are presented in Fig. 5.6(b), giving an approximated function size of 22.

```
return_type
fun_name(param_list) {
  function_body;
}
```

Function definition's size approximation →

return type size
function name size
parameter list size
function body size
function weight +
function size

Fig 5.6 (a) function definition construct size approximation

```
void sum (int t){
 if(t == 0)
   return 10;
 else
 return
   (t + sum (t-1));
}
```

Function size approximation →

return type:	1
function name:	1
parameter list:	2
if condition:	3
if then statement:	2
if else statement:	8
function weight:	5 +
total function size: 22	

Fig 5.6(b) Function definition size approximation

FIG. 5.6 FUNCTION-SIZE APPROXIMATION

A non-terminal node's weight is used to make some possible adjustments. For example, by invoking a function, an activation record will be constructed and parameters will be passed. This generally takes more time and instructions than calling a floating-point addition instruction, so a function weight (value 5) is assigned to each

FunctionCallSiteNode. Actually, all terminal nodes also have weights (not size), which have been presumably assigned value 0 and need not appear in the size evaluation.

The experience gained from our inliner implementation shows that this size approximation is good enough, because both the source-code size and the size of anticipated inlining growth are going to be approximated in the same way. Accurate estimations are not critically important.

5.4.2 The Version Issue

There might be many versions of each function during inlining, but only one version will be used at a particular inlining step. For instance, an arc a = ($\mathbf{P}_i$,$\mathbf{P}_j$) is to be inlined. After inlining the callee $\mathbf{P}_j$ with its original version, the callsite is replaced with the duplicated function body and proper adjustments. Theoretically speaking, it is semantically equivalent to replace callee $\mathbf{P}_j$ with any possible version of $\mathbf{P}_j$ when inlining, although they might have significant operational differences. Actually, it is useful to distinguish two different versions of each function.

- ov – original version

The program before any inlining is made is used to duplicate function body.

- cv – current version

The current callee is used to duplicate function body.

As described in [7,13], it is prohibitively expensive to maintain all versions as the inliner is progressing. Thus, only the original version and current version of each function will be maintained during inlining. Note also that the inlining of any function's current version can be simulated by an appropriate sequence of original-version inlinings.

An important difference between cv-inlining and ov-inlining at a callsite is that ov-inlining leads to smallest code expansion, but not always lead to fewer function calls. Ov-inlining minimizes code expansion because we assume inlining always makes functions larger and so the current version of the callee is never smaller than its original version. However, one ov-inlining might actually lead to more calls, as shown in Fig. 5.7. Its left part shows some source code, and the right part shows the result after one ov-inlining step. Next, there is a possibility that the callsite to *f* in *main* could be either ov-inlined or cv-inlined. If it were ov-inlined, there would actually be 1000000 - 1 more calls, because all the calls to *g* in the original version of *f* come back in this situation.

```
g(){
  /* g's work */ };
f(){
   /* f's work */
   a loop iterates 100000 times
        call g();
  }
 main(){
  call f();
 }
```

Source program before inlining

ov-inlining on callsite g →

```
g(){
  /* g's work */ }
f(){
  /* f's work */
  a loop iterates 100000 times
  { /* inlined code for g() */ }
   }
 main(){
      call f();
 }
```

ov-inlined program

FIG. 5.7 OPERATIONAL DIFFERENCE ON VERSION ISSUE

On the other hand, one cv-inlining step could lead to fewer function calls at the expense of a larger code expansion.

5.4.3 Hybrid Inlining Strategy

In this section, we describe a greedy strategy that aims to remove as many callsites as possible given a limited code expansion. Knowing that ov-inlining always leads to smaller code expansion and cv-inlining always leads to fewer calls, our strategy considers both cases for each callsite. The description of the hybrid algorithm is adapted from [11] and presented in Fig. 5.8. Since this implementation modifies the algorithm to support multiple techniques, the calculation of benefit is significant.

Symbols used in Fig. 5.8 include

σ_{ov} (**P**): original-version function-size of routine **P**

σ_{cv} (**P**): current-version function-size of routine **P**

5.4.4 Explanation

The whole inlining processing consists of five main steps

- Benefit Calculation

For each callsite arc a = ($\mathbf{P}_{i,}\mathbf{P}_{j}$) in the current call graph, both ov-inlining and cv-inlining effects are calculated. The details of the formulas used can be found in [11,13]. For each type of inlining, values need to be calculated such as cost, benefit and ratio, where ratio is defined as benefit over cost.

- Inlining Decision

The critical factor for the inlining decision is the cost/benefit ratio that determines which callsite is inlined and which form of inlining (cv-inlining or ov-inlining) is used. The algorithm scans the entire call graph, trying to find the callsite to inline (which is recorded by "best"). After the scan, "best" indicates if there is a callsite that needs to be inlined, and "bestKind" records the inlining type.

```
currentExpansion = 0
while (1) { /* find the best inlining operation */
  best = 0;
  for each arc a = (P_i, Pj) in the current call graph
  {
     ovBenefit = reduction in calls, if a is ov-inlined
     ovCost  = σ_ov (Pj);
     ovRatio = ovBenefit/ovCost;

     cvBenefit = reduction in calls, if a is cv-inlined
     cvCost  = σ_cv (P_j);
     cvRatio = cvBenefit/cvCost;
     if(constant propagation possible) { cvRatio = cvRatio * 1.1;
                                         ovRatio = ovRatio * 1.1; }
     if(parallel callsites in the enclosing loop) { cvRatio = cvRatio * 0.5;
                                                    ovRatio = ovRatio * 0.5; }
      if best < cvRatio then  {
         best = cvRatio;
         bestKind = cv;
        }
      if best < ovRatio then  {
        best = ovRatio;
        bestKind = ov;
        }
   } /* end of each callsite consideration */

    if (best # 0)           exit program
    if (bestKind == cv)     { currentExpansion = currentExpansion + σ_cv (P_j);
                                perform ov-inlining;  }
                        else  {  currentExpansion = currentExpansion + σ_ov (P_j);
                                perform cv-inlining;   }
     update σ_cv (P_i)
     update parse tree and current call graph
     update data structures used to calculate ovBenefit and cvBenefit

 } /* end of infinite loop */
```

FIG. 5.8 PSEUDOCODE FOR AN INLINING-DECISION ALGORITHM, BASED ON FUNCTION-CALL REDUCTION

• Inlining Work

The actual inlining work then needs to be performed on the chosen callsite, using the code transformation techniques introduced in Chapter 4. After inlining the selected callsite, it is necessary to update both the parse tree and the current call graph to reflect the current inlining effects.

• Update Accompanying Data Structure

"CurrentExpansion" is updated immediately after inlining to reflect the increased code size. Since we model the inlined function's size as the sum of caller's size and callee's size, "currentExpansion" is adjusted accordingly. Note that not only the call graph and the parse tree, but also some accompanying data structures need to be updated. These data structures permit efficient calculation of ov-benefit and cv-benefit, which is a determining factor for making inlining decisions. The updating of the data structures is done exactly as described in [11] and hence details are omitted.

• Inlining Termination

Note that we quit inlining when "best" is non-positive. This occurs when the decision algorithm scans the entire call graph and does not find any callsite worth inlining. Thus the whole inlining process terminates.

5.4.5 Cache Effects

As discussed in [12], inlining plays a trade-off between cache size and cache misses. On one hand, inlining always leads to removal of expensive call and return instructions and improves code locality, which is beneficial for cache performance. On the other hand, it also is possible that the expanded code could increase the chances of cache misses. A significant negative behavior thus could appear if the increased misses were experienced in the body of a loop. We next consider several cases and explain how our inlining decisions can be based on cache issues.

- Single Callsite Outside of Loop

If a callsite located in a function and statically outside of any loop is to be inlined, the callee's function size is relatively unimportant. If this size is smaller than the cache size, inlining will have a definite benefit: removal of a pair of expensive call-return instruction without increasing the cache misses. However, if the function's size is greater than the cache size, it is hard to say whether the inlined code would still improve cache performance except for the eliminated instructions, because both versions have cache misses. Depending on the actual function size, cache characteristics and code replacement strategy, it is even harder to analyze which version (before inlining or after inlining) would have more misses. Hence if the callsite is outside of any loop and recursion is not allowed, performance effect after inlining is negligible. Previous research [6,12] shows that it is not guaranteed that cache misses would increase and it might be better to inline in this case.

- Callsite Inside Loops

Both statistical analysis of callsite locations and practical programs [7,10] show that it is very likely that a given callsite will be located inside a loop. As McFarling [12] showed, inlining effects within loop are more complicated than the previous case. An example is given in Fig. 5.9, where circles represent loops, letters represent callsite names and arcs represent nesting.

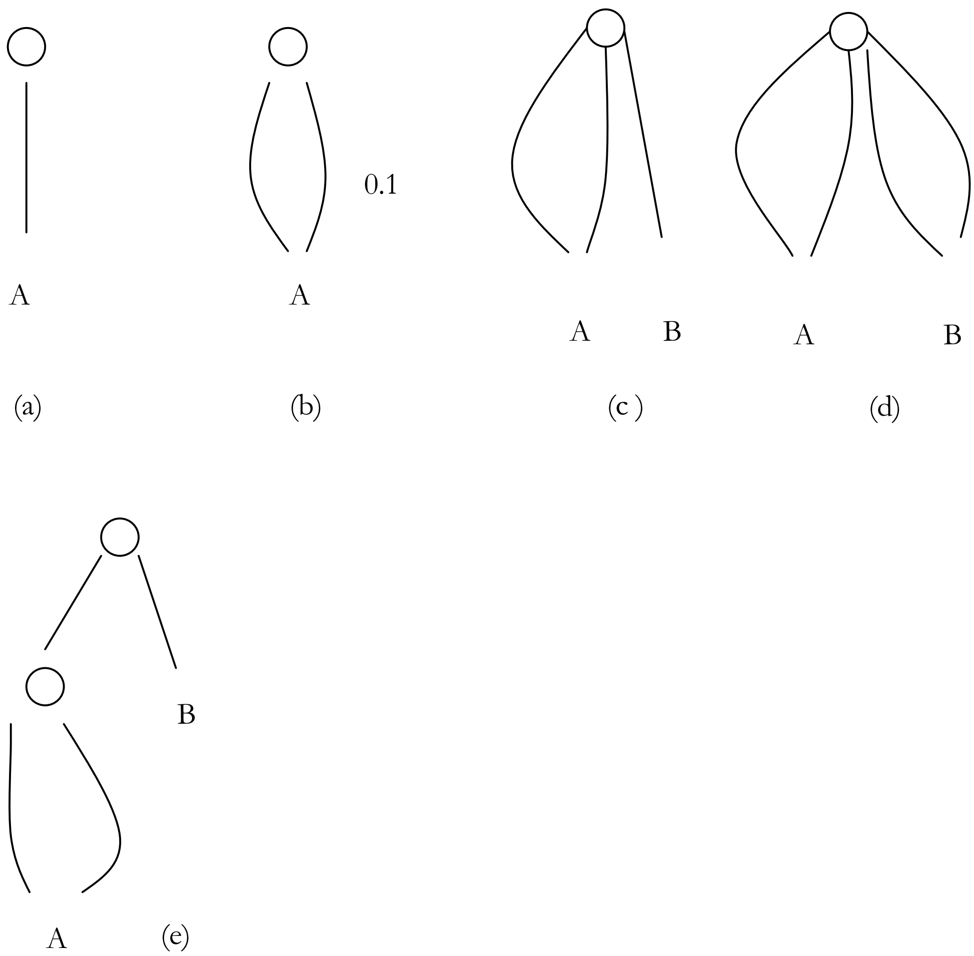

FIG. 5.9 CALLSITE INLINING IN LOOPS

Fig. 5.9(a) shows a function called from a single site inside a loop. The effect is similar to inlining a single callsite outside of any loop, a situation discussed earlier. Fig. 5.9(b) shows the same function called from different sites in the same loop, one frequently called and the other rarely executed. In case we need to inline a callsite in this situation, it seems to be more useful to inline the callsite with higher execution frequency because the higher reduction of call and return instructions could possibly compensate more for the increased misses. However, after inlining of the first callsite, it seems that the second callsite can be inlined for free, because the copy of the caller can replace the callee itself in the cache. This is one more benefit that inlining callsites in loop might have. Consider a case where the size of the loop almost perfectly matches the size of cache (maybe the loop is a slightly smaller). After inlining either of the callsites inside the loop, there might be increased cache misses in each iteration of the loop. The increased misses could be important, depending on the loop size, free code-space left in cache, cache replacement algorithm, function's dynamic behavior and the number of iterations the loop performs. If the accumulated cache misses happen to do more harm than the benefit gained from instruction removal, the inlining will actually make the program run slower. This violates the goal of inlining and should be avoided.

Fig. 5.9 (c) shows a situation where function A and B are both located in the same loop. The difference is that callsite A is invoked at two different sites and there is only one place to invoke function B. From the former analysis of a single callsite inside a loop, we know it is better to inline callsite B first, due to the elimination of instructions and overall reduction on loop size. Whether callsite A should be inlined depends on the size of callee function A and the space left in cache after inlining callsite B. Apparently, if A is small enough that after inlining there will be still place left in cache, then callsite A should be inlined. Otherwise, if A is relatively large and cache is already full, inlining should be avoided because the increment in loop size will cause more cache misses in each iteration.

Fig. 5.9(d) shows the situation where several callsites invoke functions from different places inside a loop. Inlining effects depend on the size of the function, the code space left in cache and the original size of the loop. It is usually beneficial to consider inlining small calls first.

Fig. 5.8(e) shows a situation where callsite A is nested deeper than callsite B. Apparently, if there is an opportunity that only one callsite could be inlined, callsite A should have higher priority, because deeply nested loops generally have more callsites (we could know this easily from profile information), hence more instructions removed after inlining. Similarly, whether or not callsite A or B could be actually inlined depends on the size of callee, the size of loop and the free space left in cache. If the loop body could still fit into cache after inlining, the inlining will have a positive result, then callsite should be inlined. Otherwise, "the additional misses must be carefully weighted against the removal of instructions in the loop"[12]. It is generally a good suggestion not to inline any callsite under such a situation.

According to the situations discussed above, there are three important factors to consider regarding cache issue. These functions are discussed, beginning with the most important.

First, very small functions should be inlined. There could be a small function whose body is even smaller than the cost of calling sequence. Inlining such function callsites should always lead to benefit.

Second, choose functions with the highest ratio of number of calls over function size. This is similar to the decision algorithm we described in Fig. 5.8. The callsite with the highest ratio (ovRatio or cvRatio) will be inlined, until a threshold is reached. This limits the code expansion to an acceptable level.

Third, choose callsites that do not have any parallel calls in the call graph. This is introduced in Fig. 5.9(a). Inlining single callsites (inside a loop or outside of a loop) reduce the number of instructions executed without clogging the cache with multiple copies of the same code.

Even after analyzing the complex situations that callsites in loops might have, it is still not clear how to make accurate cache predictions. We have the general idea that if the loop's dynamic size is less than cache size after inlining, the inlined benefit is the removal of call and return instructions plus some reductions in cache misses. On the other hand, if the routine size is greater than cache size, the inlined code still has cache misses because the original one already had them. And it is hard to say whether the misses will increase or not after inlining. Furthermore, whether the increased cache misses will compensate for the benefit gained from instruction elimination is difficult to predict.

Davidson and Holler [5,6] found that "the size of an inlined program does not, in practice, prove to diminish the program's performance". Their results were mainly based on testing a large set of programs and doing statistical analysis, rather than detailed theoretical analysis and proofs. Richardson and Ganapathi [20,21] found similar results on different architectures. Different machine architectures have difference cache systems. Considering cache would make the inliner optimize for a specific architecture thus would lose more on other architectures and break the idea of "generic inlining optimization".

Recall that the primary benefit of inlining is the removal of expensive call-return instructions. Cache analysis does show that there are chances that combining cache factor in some ideal invocation (e.g. in loops) could benefit the inliner. However, the time limitation on inliner implementation makes it impossible to carefully combine cache factors and successfully integrate into the inliner. Based on this analysis, we believe that cache consideration is relatively unimportant in the inliner's implementation.

5.4.6 Recursive Function Inlining

Besides the label-renaming scan introduced in Chapter 4, recursive functions are treated the same way as non-recursive ones, except that the cvBenefit is evaluated differently, as described in [11,13]. Actually, for a recursive callsite, we are more likely to perform ov-inlining. This *may* lead to more call and return instructions removed, as shown in Fig. 5.10, which is adapted from [11].

```
void f(int x){
 if (x == 1) return 0;
 else {   y = f(x-1);
          return 1 –y;   }
  }
…
main(){…
   f (300);
    …  }
```

FIG. 5.10 EXAMPLE OF MORE CALLSITES' REMOVAL WITH OV-INLINING

Consider the recursive function above. Clearly if the original version of callsite *f* is inlined twice, it leaves us the program where *f(x)* invokes *f(x-3)* with *f* now thrice its original size. If this is the maximum code expansion permitted, cv-inlining is able to inline once at the recursive site, because a second cv inlining will leave *f* at quadruple size, which has exceeded the maximum expansion. Thus ov-inlining removes 50% more calls than cv-inlining in this example.

5.4.7 Decision on Constant Propagation

The decision algorithm explicitly checks whether there is a constant-propagation opportunity for some callsites with constant parameters. Details have been given in

Section 4.5. If any constant-propagation opportunity is detected to be possible on a callsite, both the ovRatio and the cvRatio will increase by a small amount (we arbitrarily choose 10%). This will give those callsites with constant propagation opportunities a higher priority to be selected for inlining.

5.4.8 Cache and Recursion

Another interesting issue is simultaneously considering cache and recursion. Consider a simple case of a directive recursive function where there is only one callsite that appears in the definition, as shown in Fig. 5.11(a). The execution behavior could be simulated as two while loops separated by an exit case, as shown in Fig. 5.11(b) and Fig. 5.11(c). The first while loop simulates the behavior of recursion when it goes deep to find its "base case". In our case, all possible recursive callsites will appear in this while loop. Whenever the recursion reaches its base case, the returned variable is assigned return value and the second loop actually deconstructs the recursive activation records and returns.

Let us assume this is the case that a recursive callsite located inside a loop (pseudo loop), which fits into Fig. 5.11(c). Now that we have theoretically converted the run-time behavior of a recursive callsite into a loop structure, the techniques introduced in Section 5.4.5 could be directly used to discuss its inlining. As described earlier, it is better to inline these callsites if the pseudo loop (the conditions before reaching the base case in the recursive definition) is not too big to fit into cache. Some conditional analysis are presented in Section 5.4.5.

```
…
sum(4); /* callsite */
…
int sum(int i){
 if(i ==0) return 0;
  else
  return i+ sum(i-1);
}
```

Fig 5.11(a) Simple directly recursive function

```
  sum(4)
→[4 + [sum(3)]]
→[4 + [3 + [sum(2)]]]
→[4 + [3 + [2 + [sum(1)]]]]
→[4 + [3 + [2 + [1 + [sum(0)]]]]]
→[4 + [3 + [2 + [1 + 0]]]]
→[4 + [3 + [2 + 1]]]
→[4 + [3 + 3]]
→[4 + 6]
→10
```

Fig 5.11(b) Example of recursive callsite trace, where […] are callsites

```
…
{
while (condition1)
  go deeper nesting in recursion
base case
while (condition2)
 return from deeply nested recursion
}
…
```

Fig 5.11(c) Pseudo code simulation for runtime behavior of recursive callsite

FIG. 5.11 SIMULATION OF RECURSIVE CALLSITE

5.7 Testing

The experiments on some test programs demonstrate the possible effects on program performance of inlining, as shown in Table 5.2 and Table 5.3.

Table 5.1 lists all of our 11 individual testing programs. The testing programs range widely from Unix system code, to publicly available commercial program, to specific user code. All testings are performed on a PentiumPro based machine, running PC-Linux kernel 2.0.29[14].

The testing consists of two parts. First, we compare standardized programs' run-time performance with that of non-standardized. This is to see what kind of effects the function callsite standardization could have on the program's execution behavior. The results are presented in Table 5.2. The second part of testing is to see the possible benefit of inlining to a program. Both program run time and the number of calls removed are calculated and compared.

Name	Description	Type
uuencode	Program encoding	Unix system
uudecode	Program decoding	Unix system
chmod	Source code for chmod command	Unix system
cp	Source code for cp command	Unix system
mv	Source code for mv command	Unix system
touch	Source code for touch command	Unix system
strcpy	Strcpy in loops	User code
nfib	Recursive fibonacci function	User code
nq	Recursive n queens problem	User code
tak	Recursive tak function	User code
sumfrom	Recursive sumfrom function	User code

Table 5.1 Programs used in inlining experiments

[14] Gcc 2.7.2.1 for Pc-linux. The command-line switch is: gcc –c source_file_name –fno-inline –o dest_file_name.

Testing program	Size before standardiza-tion (bytes)	Run time before standardization (sec)	Size after standardization (bytes)	Run time after standardization (sec)	Performance change (%)
uuencode	36832	0.048709	37140	0.049575	1.78
uudecode	39079	0.040830	39287	0.041667	2.05
chmod	92593	0.018346	92671	0.018583	1.29
cp	140194	0.254754	141806	0.257933	1.43
mv	111634	0.020815	112922	0.019754	-5.10
touch	110958	0.185062	111218	0.18463	-0.24
strcpy	4598	3.729231	4646	3.675381	-1.444

Table 5.2 Standardization Comparison

From Table 5.2, we see that there is very little performance effect, comparing the run-time performance of standardized programs with that of non-standardized ones (generally ranging from 1.29% to 2.05%). The reason is simple. By performing the function callsites' standardization, there are variables mandatorily created. These variables consume run-time resources. Hopefully, the negative effects are so small that they could usually be ignored.

Program	Strcpy	Nfib	Nq	Tak	sumfrom
Number of Calls before inlining	130001	1664078	1875813	965432	4005001
Number of calls after inlining	120001	1149850	1079220	813139	2685683
Number of calls removed	10000	514228	796593	152293	1319318
Percentage of callsites removed	7.69	30.9	42.47	15.77	32.94

(%)					
Execution time before inlining (second)	0.750922	0.761328	3.01811	0.407131	1.895180
Execution time after inlining (second)	0.723072	0.620637	2.67497	0.363899	1.567807
Performance Improved (%)	3.71	18.48	11.37	10.63	17.28

Table 5.3(a) Inlining Performance (20% code expansion)

Program	Strcpy	nfib	nq	Tak	sumfrom
Number of Call before inlining	130001	1664078	1875813	965432	4005001
Number of Calls after inlining	100001	832038	1010058	685946	1366365
Number of calls removed	30000	832040	865755	279486	2638636
Percentage of callsites removed (%)	23.07	50.00	46.15	28.95	65.88
Execution time before inlining (sec)	0.750922	0.761328	3.01811	0.407131	1.895180
Execution time after inlining (sec)	0.723181	0.591934	2.38779	0.327155	1.126171
Performance improved (%)	3.69	22.25	20.88	19.64	40.58

Table 5.3(b) Inlining Performance (50% code expansion)

Program	strcpy	nfib	nq	Tak	sumfrom
Number of Calls before inlining	130001	1664078	1875813	965432	4005001
Number of Calls after inlining	90001	542144	1013970	508640	1621620
Number of calls removed	40000	1121934	861843	456792	2383381
Percentage of callsites removed	30.77	67.42	45.95	47.31	59.51

(%)					
Execution time before inlining (sec)	0.750922	0.761328	3.01811	0.407131	1.895180
Execution time after inlining (sec)	0.691493	0.562159	2.38538	0.275651	1.089123
Performance improved (%)	7.91	26.16	20.96	32.29	42.53

Table 5.3(c) Inlining Performance (100% code expansion)

From Table 5.3, we see that a significant percentage of calls (8.33% ~ 67.42%) are removed after inlining. Further, performance after inlining generally increases ranging from 3.69% ~ 42.53% on some heavily computational and recursive programs.

From the experimental results, we see that the standardization process has an almost negligible effect on the program size and on the run-time efficiency. Further, based on our testing results, we see that a relatively large number of callsites were removed and this leads to a large improvement in the programs' execution efficiency after inlining. This is mainly because that the selected testing programs are heavily computational oriented. They intensively used many callsites and involved heavy recursion. For those programs that are input/output dominated or do not execute many calls, the performance improvements after inlining would be less substantial.

Chapter 6

Summary and Future Work

This chapter summarizes the contribution of this thesis and gives possible directions for future work.

6.1 Summary

In this thesis, we have developed an inlining system containing a lexical analyzer and an object-oriented parser for the C programming language. Every C construct corresponds to a parse-node class, which matches the syntactical characteristics of the construct. We parse source code separately, generate our own abstract parse tree and manage the symbol tables in a non-traditional fashion – using a dynamic array of Tree pointers. Object-oriented design patterns are used to direct the parser design and help to do the work of inlining. Most important, we have fully implemented a greedy hybrid inlining-decision algorithm, and it considers multiple versions and limits on code expansion.

In Chapter 3, we showed the detailed design of the object-oriented parser. We presented the entire parse-node class hierarchy and the design patterns that helped to reduce coding complexity and improve program quality. Important parsing and class-design techniques, such as the message system, class macro/micro architecture, reparsing and standard interfaces were also introduced.

In Chapter 4, we gave the details of the implementation of inlining. Function callsite standardization was an important issue. A series of standardization processes worked on parse tree in a leftmost-outermost order to convert any callsite into a standardized format. One important opportunity created by inlining was constant propagation, whose implementation details were also discussed.

In Chapter 5, we presented the details of the hybrid inlining-decision algorithm. Call-graph generation, removal of invalid callsites, generation of profile information and approximation of functions' sizes were introduced. Detailed analysis and explanations were also given on cache issues. After careful consideration of possible cache benefits and penalties, especially in loop situations, we showed that it was difficult to take cache characteristics into account and optimize the inlining decision algorithm on cache issue. Program-execution performance comparisons and analysis were also presented in this chapter.

6.2 Conclusion

In the thesis implementation, we have created an ambitious inliner – an automatic C inliner. It has several features that, as far as we know, do not exist in other inliners.

- Object-oriented parser.

Based on the experience we gained from the parser design, an object-oriented parser is more suitable when the optimizations to be performed do not involve too much analysis. The parse-node class design describes the syntactical constructs better and clearly simulates the inlining optimization behavior. However, compared with traditional parsers, the object-oriented parser has higher memory consumption and generally runs slower.

Its usage will be limited when the system is very sensitive to memory consumption, and it is not ideally suited to process huge programs.

• Inlining Optimization

Inlining and the optimization opportunities thus created are some of the important compiler optimization techniques and are carefully studied in the thesis. Practical results demonstrate a relatively large number of calls removed and an acceptable performance increment. As only one of the many optimization techniques, the performance increment does not need to be huge.

• Use of Design Patterns

Design patterns are powerful tools that could significantly simplify design and improve implementation quality. However, if they are improperly analyzed and used, the result can be disastrous. The careful selection and properly implementation of design patterns is important.

6.3 Future Work

There are a number of areas for possible future work that would extend the current thesis implementation.

6.3.1 Multi-File Support

Since most significant C programs consist more than one module (separated into several files), different naming and searching methods need to be used when constructing the parse tree and performing the function-name searching and function-body duplication.

Since file names will never be duplicated in the source program if the file name comes with its complete path name, they could be used as prefixes to generate distinct names for global variables and functions. Details of how to handle certain items, such as static functions and external references, would have to be carefully considered.

6.3.2 Pipeline Interlocks

As shown in [20], there could be an unexpected side effect when inlining callsites under a special assignment condition, where a variable serves as both the parameter and the result. Even worse, the same variable could appear several times in arguments, where it mandatorily creates aliases, as shown in Fig. 6.1. A certain data dependency created by alias will actually make the program run slower after inlining. (E.g., in the inlined version of the Fig. 6.1 example, there will have many NOP instructions inserted after modifying the simulated callsite arguments that is to properly maintain the data dependency.) The increased number of pipeline interlocks caused after inlining will compensate more than the benefit gained by inlining, thus inlining in this situation is not valuable. This situation is generally common in some popular pipelined microprocessor systems and currently not considered in our thesis. This hazardous situation should be carefully detected and callsites under this situation should be prohibited inlining.

```
…
 a[10] = f(a[10],a[10]);
 …
int f(int i, int j){
 i++; j--;  /* code of function A */
return (i+j);
}
```

FIG. 6.1 EXAMPLE OF PIPELINE INTERLOCKS

6.3.3. Database in Inlining

Dean and Chambers [9] considered inlining in an object-oriented language where there is a similarity based on callsites signatures (return type, callsite name, and type of parameter list, especially class types). A database could be used to record the function signature, function-calling frequency and compiler-optimization opportunities obtained from one inlining step. This could help to reduce the inlining-decision time and simplify the decision-making algorithm.

References

[1] Flex lexical analyzer generator v 2.91, Free Software Foundation, last updated at 09/10/96

[2] Berkeley Yacc Parser Generator, University of California at Berkeley, last update at: 01/30/90

[3] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad and Michael Stal, Pattern-Oriented Software Architecture, John Wiley & Sons, 1996

[4] American National Standard Institute C grammar, Comp.compilers Archie site located at: http://iecc.com/cgi-bin/getarticle?91-09-030 ANSI 1988

[5] Jack W. Davidson and Anne M. Holler, A Study of a C Function Inliner. Software – Practice and Experience, 18:775-790, 1988

[6] Jack W. Davidson and Anne M. Holler. Subprogram Inlining: A Study of its Effects on Program Execution Time. IEEE Transactions on Software Engineering, Vol. 18, No.2, Feb. 1992.

[7] Pohua P. Chang, Scott A. Mahlke, William Y. Chen and Wen-mei W. Hwu. Profile-Guided Automatic Inline Expansion for C Programs. Software – Practice and Experience, 25:349-369, 1992

[8] Keith D. Cooper, Mary W. Hall and Linda Torczon. Unexpected Side Effects of Inline Substitution: A case study. ACM Letters on Programming Languages and Systems, 1:22-32, 1992

[9] Jeffery Dean and Craig Chambers. Towards Better Inlining Decisions Using Inlining Trials. ACM Conference on LISP and Functional Programming, Orlando, Fl, June 1994

[10] Wen-mei W. Hwu and Pohua P. Chang. Inline Function Expansion for Compiling C Programs. The SIGPLAN '89 Conference on Programming Language Design and Implementation, pages 246-255, 1989

[11] Owen Kaser and C.R. Ramakrishnan. Evaluating Inlining Techniques. Technical Report TR 96-001, Dept of Math, Stat and Computer Science, U. of New Brunswick, Saint John, Canada, Aug 1996. Scheduled to appear in Programming Languages, 1998

[12] Scott McFarling. Procedure Merging with Instruction Caches. Proceedings of the SIGPLAN' 91 Conference on Programming Language Design and Implementation, ACM, 1991

[13] Robert W. Scheifler. An Analysis of Inline Substitution for a Structured Programming Language. Communications of the ACM, 20(9): 647-654, 1977.

[14] Graham, S.L., Kessler, P.B., McKusick, M.K., `gprof: A Call Graph Execution Profiler, Proceedings of the SIGPLAN '82 Symposium on Compiler Construction, SIGPLAN Notices, Vol. 17, No. 6, pp. 120-126, June 1982.
http://www.cs.ubc.ca/local/software/GNU_info (accessible April 1998)

[15] Minda Zhang, Analysis of Object File Formats for Embedded Systems, Intel Corporation, June 1995.
http://developer.intel.com/design/intarch/PAPERS/ESC_FILE.HTM (accessible April 1998)

[16] Executable Language Specification, Intel Corporation
ftp://ftp.intel.com/pub/tis/elf11g.zip (accessible March, 1998)

[17] Keith D. Cooper, Mary W. Hall and Linda Torczon, Unexpected Side Effects on Inline Substitution: A Case Study, ACM Letters on Programming Languages and Systems. Vol. 1, No. 1, March 1992. Pages 22-32

[18] Rinus Plasmeijer and Marko van Eekelen, Functional Programming and Parallel Graph Rewriting, Addison-Wesley, 1993

[19] S. Richardson and M. Ganapathi. Interprocedural analysis vs. procedure integration. Information Processing Letters, 32(3): 137-142, August 1989

[20] S. Richardson and M. Ganapathi. Interprocedural optimization: Experimental results. Software —Practice and Experience, 19(2): 149-169, February 1989

[21] Alfred V. Aho, Ravi Sethi and Jeffrey D. Ullman, COMPILERS Principles, Techniques and Tools, Addison Wesley Publishing Company, 1986.

[22] HP Compiler Optimization White Paper, Hewlett Packard Corporation. November 1996
http://www2.hp.com/wsg/ssa/fortran/optimiz.html

[23] Owen Kaser, C.R. Ramakrishnan, I.V. Ramakrishnan, R.C. Sekar, EQUALS – A Fast Parallel Implementation of a Lazy Language, Journal of Functional Programming, Vol. 7 No. 2, 1997.

[24] GNU C++ language compiler package, version 2.7.2.1, Free Software Foundation. Ftp site at http://www.cs.ubc.ca/local/software/GNU_info gnu Canadian ftp mirror.

[25] Linux operating system package for PC version 5.0, kernel 2.0.29, RedHat Software Incorporated. Ftp download site at http://www.redhat.com/ (installed in May 1997)

[26] Intel C/C++ Compiler White Paper, Intel Corporation.
Http site located at http://134.134.214.2/design/perftool/icl24/icl24wht.htm (accessible April 1998)

[27] Unified Modeling Language standard published by Rational Rose Software.
Http site located at http://www.rational.com/uml/html/semantics/semanta1.html (accessible April 1998)

Appendix

Appendix 1 Parse-Node Internal Data Structures

Class Name	C Construct	Example	Internal Data Structure
String node	Character string	"hello", val1;	char * StringValue;
Double node	Double value	3.1415926	double DoubleValue;
Char node	Character value	'c'	char CharValue;
Long node	Long int value	100L	long LongValue;
Integer node	Int value	123	int IntValue;

Table 1-1 Terminal leaf nodes

Class Name	C Construct	Example	Internal Data Structure
BinaryNode	Binary operations Unary operations	1+b, c && t, t++	Tree * Left; Tree * Operation; Tree * Right;
PointerNode	Pointer operations	r->s, r->hello( );	Tree * PointerName; Tree * ReturnType; Tree * PtrInit;
ArrayNode	Array definition Array operations	int a[100] = {0}; a[i] = a[i+1];	Tree * ArrayType; Tree * ArrayName; Tree * ArrayInit;
EnumNode	Enumerations	enum color {red = 1, green , blue = 3};	Tree * EnumName; Tree * EnumInit;
Struct/Union Node	Struct/union declarations	struct { int i; float r; } mystruct;	Tree * Name; Tree * Body; int Type;
JumpNode	goto, return, break statements	goto label_1; return a + f(x);	int JumpType; Tree * JumpExpr;
SwitchCaseNode	Switch_case control	switch(f(x)){ statement;}	Tree * SwitchCond; Tree * SwitchStmt;

Table 1-2. Non-terminal leaf nodes

Class Name	C Construct	Example	Internal Data Structure
IfNode	if_then_else control	if(condition) then stmt; else stmt;	Tree * Condition; Tree * ThenTree; Tree * ElseTree;
ForNode	for conditional control loop	for(i=0;i<10;i++) printf(“%d\n”,i);	Tree * Initializer; Tree * Finalizer; Tree * Modifier; Tree * LoopBody;

Table 1-2 Non-terminal nodes (continue)

Class Name	C Structure	Examples	Internal Data structure
DoWhileNode	do_while conditional control loop	do{ i++; } while (i< 100);	Tree * LoopBody; Tree * Condition;
WhileNode	While conditional control loop	while (i <= 100) { i++; }	Tree * Condition; Tree * LoopBody;
LabelNode	C labels	Exit1: I = 5;	Tree * Label; Tree * LabelStmt;
BracketNode	{ … }	{ int i; … i = 10; … }	Tree * DeclTree; Tree * StmtTree;
LST	Local symbol table declarations	int i; float j = f(i);	Tree ** LSTChain
FunctionCallSite Node	Function call site	f(x); f1(1+b);	Tree * FunName; Tree * ParamList;
FunctionNode	Function declaration	int f(int x){ … }	Tree * ReturnType; Tree * Name; Tree * ParamList; Tree * FunBody;
FunctionPrototype Node	Function prototype declaration	int f(int,int,int); float sqr(float);	Tree * ReturnType; Tree * Name; Tree * ParamList;
CallSiteElement Node	Function call site chain	f(x) … f(1) … f(t) …	Tree * CallSiteChain
FunctionCallSiteList Node	function call site element chain (collection of all function call sites)	f1(x)… f2(x)… f3(x)…	Tree * CallListChain;

Table 1-3 Complete Parse-Node Data Structure

Class Name	C Structure	Examples	Internal Data structure
TypeNode	C predefined type declaration	int, long, char, float, double, sizeof, static, extern, register, …	int Type;
MacroNode	line directive declaration	# line 333 "gram.y"	int linenumber; char * filename;

Table 1-3 Complete Parse-Node Data Structure (continue)

Appendix 2 Memory Consumption of Parse Nodes

Class Name	Memory Consumption (Bytes)
Integer Node	44
Long Node	46
Char Node	45
Double Node	50
String Node	56
Type Node	44
Binary Node	68
Bracket Node	52
Symbol Table Node	44
LST	44
Struct Union Node	56
Enumeration Node	50
Array Node	58
IfThenElse Node	60
SwitchCase Node	56
For Loop Node	64
While Node	56
DoWhile Node	56
Function Node	66
FunctionPrototype Node	58
FunctionCallSite Node	130
Label Node	52
CMessage	69
Macro Node	46

Appendix 3 Parse-Node Size Approximation

Class Name	Weight	Size Approximation
String Node	0	1
Integer Node	0	1
Char Node	0	1
Double Node	0	1
Long Node	0	1
Binary Node	0	Left Size + Operation Size + Right Size + Weight
Pointer Node	0	Name Size + Content Size + Initialization Size + Weight
Array Node	0	Name Size + Index Size + Initialization Size + Weight
Enumeration Node	0	Name Size + Enumeration List Size + Weight
Struct Union Node	0	Name Size + Declaration Size + Weight
Jump Node	0	Jump Expression Size + Weight
SwitchCase Node	3	SwitchCase Condition Size + SwitchCase Body Size + Weight
IfThenElse Node	3	IfThenElse Condition Size + Then Clause Size + Else Clause Size + Weight
For Node	3	Initializer Size + Finalizer Size + Modifier Size + Loop Body Size + Weight
While Node	3	While Condition Size + While Loop Body Size + Weight
Do While Node	3	Do While Condition Size + Do While Loop Body Size + Weight
Label Node	0	Label Size + Label Expression Size + Weight
Bracket Node	0	Declaration Size + Statement Size + Weight
Local Symbol Table Node	0	Summary of Local Symbol Size + Weight
Function Call Site Node	0	Function Name Size + Function Parameter List Size + Weight
Function Node	5	Function Return Type Size + Function Name Size + Function Parameter List Size + Function Body Size + Weight
Type Node	0	1
Macro Node	0	2

Appendix 4 Message Details

No	Message Name	Message Value	Action Message Perform
1	ck_SetEnumIdentifier	ck_Class_Msg + 2	Set enumeration construct name
2	ck_SetEnumList	ck_Class_Msg + 6	Set enumeration content Tree pointer
3	ck_SetStructUnionHead-Name	ck_Class_Msg + 10	Set struct/union definition name
4	ck_SetStructUnionTail-Name	ck_Class_Msg + 12	Set struct/union definition alias name
5	ck_SetStructUnionBody	ck_Class_Msg + 14	Set struct/union definition body content
6	ck_SetArrayReturnType-Name	ck_Class_Msg + 26	Set array variable or definition return type
7	ck_SetArrayName	ck_Class_Msg + 28	Set array name
8	ck_SetArrayIndex	ck_Class_Msg + 30	Set array index
9	ck_SetArrayInitialization	ck_Class_Msg + 32	Set array initialization
10	ck_SetPointerReturnTypeName	ck_Class_Msg + 42	Set pointer return type
11	ck_SetPointerName	ck_Class_Msg + 44	Set pointer (definition or variable) name
12	ck_SetPointerInitialization	ck_Class_Msg + 46	Set pointer initialization
13	ck_SetLabelNodeLabel	ck_Class_Msg + 64	Set label identifier
14	ck_SetLabelNodeLabel-Statement	ck_Class_Msg + 66	Set label statement
15	ck_SetIfThenElseCondition	ck_Class_Msg + 74	Set if_then_else construct condition
16	ck_SetIfThenElseThenTree	ck_Class_Msg + 78	Set if_then_else construct then statement
17	ck_SetIfThenElseElseTree	ck_Class_Msg + 82	Set if_then_else construct else statement
18	ck_SetSwitchCaseNode-Condition	ck_Class_Msg + 86	Set switch_case construct condition
19	ck_SetSwitchCaseNode-Expression	ck_Class_Msg + 90	Set switch_case construct statement
20	ck_SetWhileNodeCondition	ck_Class_Msg + 94	Set while loop construct loop condition

21	ck_SetWhileNodeLoopBody	ck_Class_Msg + 98	Set while construct loop body
22	ck_SetDoWhileNode-Condition	ck_Class_Msg + 102	Set do_while construct loop condition
23	ck_SetDoWhileNodeLoop-Body	ck_Class_Msg + 104	Set do_while construct loop body
24	ck_SetForNodeCondition-Initializer	ck_Class_Msg + 110	Set for loop initializer
25	ck_SetForNodeCondition-Finalizer	ck_Class_Msg + 112	Set for loop condition
26	ck_SetForNodeCondition-Modifier	ck_Class_Msg + 114	Set for loop modifier
27	ck_SetForNodeLoopBody	ck_Class_Msg + 116	Set for loop body
28	ck_SetFunctionPrototype-ReturnTypeName	ck_Class_Msg + 158	Set function prototype definition return type
29	ck_SetFunctionPrototype-Name	ck_Class_Msg + 162	Set function prototype definition name
30	ck_InsertPrototypeParam-List	ck_Class_Msg + 170	Set function prototype definition parameter list
31	ck_SetFunctionReturnTypeName	ck_Class_Msg + 178	Set function definition function return type
32	ck_SetFunctionName	ck_Class_Msg + 182	Set function definition function name
33	ck_SetFunctionParameter-List	ck_Class_Msg + 186	Set function definition parameter list
34	ck_SetFunctionBody	ck_Class_Msg + 184	Set function definition body context
35	ck_SetFunctionCallSite-FunctionName	ck_Class_Msg + 202	Set function callsite name
36	ck_SetFunctionCallSite-ParameterList	ck_Class_Msg + 206	Set function callsite parameter list
37	ck_FunctionClone	ck_Class_Msg + 214	Clone function definition
38	ck_SetFunctionEqual	ck_Class_Msg + 216	Check function equal
39	ck_SetTypeNodeType	ck_Class_Msg + 224	Set C predefined data type
40	ck_SetParent	ck_Class_Msg + 226	Set current Tree pointer's parent Tree pointer
41	ck_SetBracketNode-DeclarationBlock	ck_Class_Msg + 252	Set curly bracket declarations
42	ck_SetBracketNode-StatementBlock	ck_Class_Msg + 256	Set curly bracket statements

43	ck_HasFunctionCallSite-Node	ck_INLINE + 2	Check if there is a callsite in the current construct
44	ck_HasNestedFunction-CallSite	ck_INLINE + 4	Check if there are nested callsites in the current callsite
45	ck_SetFunctionCallSiteInline Able	ck_INLINE + 8	Mark a callsite to be inlineable
46	ck_SwapAvailConditionSite	ck_INLINE + 12	Swap callsites in conditional control constructs
47	ck_SwapAvailDeclSite	ck_INLINE + 13	Swap between declarations and statements
48	ck_SplitFunctionCallSite-Node	ck_INLINE + 16	Split nested callsites
49	ck_SetFunctionCallSiteNodeS plit	ck_INLINE + 20	Split one-level callsite parameter
50	ck_RemoveFromFunction-CallList	ck_INLINE + 30	Remove uninlineable callsite from callsite list
51	ck_ReplaceCommas	ck_INLINE + 32	Comma operator removal
52	ck_DecideInlining	ck_INLINE + 40	Inlining decision algorithm
53	ck_DoRealInline	ck_INLINE + 42	Performing Inlining
54	ck_ProcessReturnClauseIn-FunctionBody	ck_INLINE + 50	Process return statement in duplicated function body
55	ck_LocalVariablePropogate	ck_INLINE + 52	Local variable renaming, Constant propagation
56	ck_SplitAndExpression	ck_INLINE + 60	Split AND expression
57	ck_SplitOrExpression	ck_INLINE + 62	Split OR expression
58	ck_RenameLabels	ck_INLINE + 76	Label renaming
59	ck_CollectFunctionNames	ck_INLINE + 78	Function callsite name collection (to build callsite candidate list)
60	ck_GetNodeSize	ck_INLINE + 82	Function size approximation

VITA

Candidate's full name:

Chengyan Zhao

Place and date of birth:

Beijing, China People's Republic of
January 04, 1973

Permanent address:

P.O. Box 927, 100083
Beijing, China P. R.

Schools attended:

Beijing Petroleum High (Junior), 1986 – 1988
Beijing Petroleum High (Senior), 1988 – 1991

Universities attended:

Peking University, Sept. 91 – July. 95
B.Sc. in Computer Science
University of New Brunswick, Master of Computer Science, Sept. 96 ~ July. 98

Publications:

1. Zhao Chengyan, *Bulletproof ATM*, Computer and Communication News, Dec. 1997.

2. Zhao Chengyan, *Bulletproof ATM*, Computer and Communication News, Nov. 1997.

3. Zhao Chengyan, *Register CHR & BGI for Borland Pascal*, MicroComputer & Applications, 33 ~ 35, Oct. 1995.

4. Zhao Chengyan, *Borland Pascal 7.0 with objects – a distinct windows development environment*, Computer and Communications, 43 ~ 44. Aug. 1995.

5. Zhao Chengyan, *Selection & installation of a 387 coprocessor*, Computer Fan, 26~27. July. 1995.

6. Zhao Chengyan, *One time totally copy of a high-density floppy diskette*, Application of Electronic Technique, 12~14. Feb. 1994.