PINLINE: A PROFILE-DRIVEN AUTOMATIC INLINER FOR THE IMPACT COMPILER

BY

BEN-CHUNG CHENG

B.S., National Taiwan University, 1992

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1996

Urbana, Illinois

ACKNOWLEDGMENTS

I would like first to thank my thesis advisor, Professor Wen-mei W. Hwu, for his guidance and support in the past couple of years. Under his instruction, I extended my view of the entire computer industry. I also would like to thank the whole IMPACT research group for their contribution to the IMPACT compiler framework. Among them, I would like to express my special gratitude to Grant Haab, who was my mentor when I was new to this group. He was always patient to answer my many questions and willing to help me. I also appreciate those sage suggestions from Dave Gallagher, Rick Hank, John Gyllenhaal and Teresa Johnson. Another important person to mention here is Brian Deitrich. His experience in implementing Linline helps me a lot in implementing Pinline, especially for handling recursive functions. I bothered him so many times to talk about the recursive issues. Without the profiling functionality added by Le-Chun Wu to Pcode, this thesis could not be possible.

Also, I would like to thank my wife, Szu-Wen, for her consistent care and encouragement. Two and a half years after her I finally get my master's degree. I wish her well toward her Ph.D. degree. In addition, my graduate life would not be complete without those many good friends in town. Without hanging around with them my school life would not be so colorful, although I could have gained the degree sooner.

Finally, I would like to thank my beloved parents. Without their love and support in every aspect, I would not be here today.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

The IMPACT C compiler was developed by the IMPACT research group [1]. It is a large, complex, but well-structured piece of software intended to exploit the highest ILP for wide-issue micro processors.

Between the C source code and the machine-dependent assembly code, the IMPACT compiler maintains three families intermediate representations: Pcode [2], Hcode [3], and Lcode [4]. Pcode is the high-level intermediate language. The grammar of Pcode maintains all constructs in C so that the Pcode intermediate representation (IR) can carry source-code information in order to perform source-to-source transformations. Pcode is further described in Chapter 2.

Hcode is the middle-level IR in the IMPACT compiler. The major difference between Hcode and Pcode is that Hcode uses flattened control structures. For example, a loop construct in Pcode is transformed into basic blocks connected via `if-then-else` and `goto` constructs in Hcode. Originally, basic-block profiling and profile-driven inlining are performed in the Hcode level. Now they have been moved into the Pcode phase.

Lcode is the low-level IR where machine-independent classic optimizations are applied. More advanced optimizing technologies like superblock and hyperblock formations are also conducted in the Lcode level [5] [6]. Then machine-dependent code generators translate Lcode into assembly code. The architectures supported by the IMPACT compiler include *AMD 29K* [7], *MIPS R3000* [8], *Sun SPARC* [9], *HP PA_RISC* and *Intel x86*.

1.1 Motivation

The invention of high-level programming languages like C facilitates the task of writing a large program by introducing the concept of modularization. For human beings, it is easier to decompose a problem into smaller modules and write code for each module. Then all the modules work in collaboration to solve the problem. In C, functions are the primary constructs for modularization. The programmers can reduce the program size by absorbing redundant operations into one function and repeatedly invoking those operations through function calls. Functions also minimize the debugging effort because the programmers can isolate the bug more easily. However, the impact of this modularization on an optimizing compiler is negative because it blocks the compiler from seeing all of the code whenever a function call is encountered. For example, it hinders the compiler from globally performing register allocation, common subexpression elimination and constant propagation [10].

Originally, the IMPACT compiler expanded function calls with Hinline [10] which operates on the Hcode level. Hinline lacks the ability to inline function pointers and recursive calls. However, the most important impetus to perform inline expansion in Pcode instead of Hcode is that Hinline is unable to properly preserve *sync arcs* [11], the memory dependence information generated by Pcode, after function inlining. In order to solve this problem we decided to perform function inlining before generating sync arcs.

1.2 Contents

The goal of this document is to give a detailed description about the Pcode inlining module, Pinline. This thesis is divided into six chapters. The next chapter will give a general overview of the IMPACT frontend modules. The third chapter will describe how the Pcode IR is flattened, so as to expose more functions to the inliner, and how the program source code is split. The fourth chapter will provide an detailed look into Pinline. The experimental results will be presented in Chapter 5 and the conclusion will be given in Chapter 6.

CHAPTER 2

OVERVIEW OF THE IMPACT FRONTEND

Conceptually, a single invocation to IMPACT will do all the compilation and optimization tasks. In reality, the IMPACT compiler is composed of many executables which either act as the bridge between two forms of intermediate representations or perform some kinds of optimizations on a specific IR level or both.

2.1 Overview of the IMPACT Parser

In the IMPACT compiler, all the modules invoked before Lcode IR is generated can be treated as the frontend modules. A C program, after being processed by the C preprocessor, is fed into an ANSI C parser called Chsemansi. It is a LALR(1)-driven parser generated by YACC. Besides parsing, it also renames static variables and inserts tags for tagless structures/unions/enumerations (referred to as structures collectively hereafter). Tags are the names of structures, like `s1` in `struct s1 { int i; }`. A FORTRAN program is first translated by F2C [12] into C code. However, F2C linearizes all arrays during translation. But with enhancement made by the IMPACT compiler group, F2C can insert delinearization information which is later translated into Pcode pragmas or directives by a PERL script called Chpp [13]. These pragmas serve an important role in the array dependence analysis performed later [13]. From that point on a FORTRAN program follows the same path as a regular C program. The

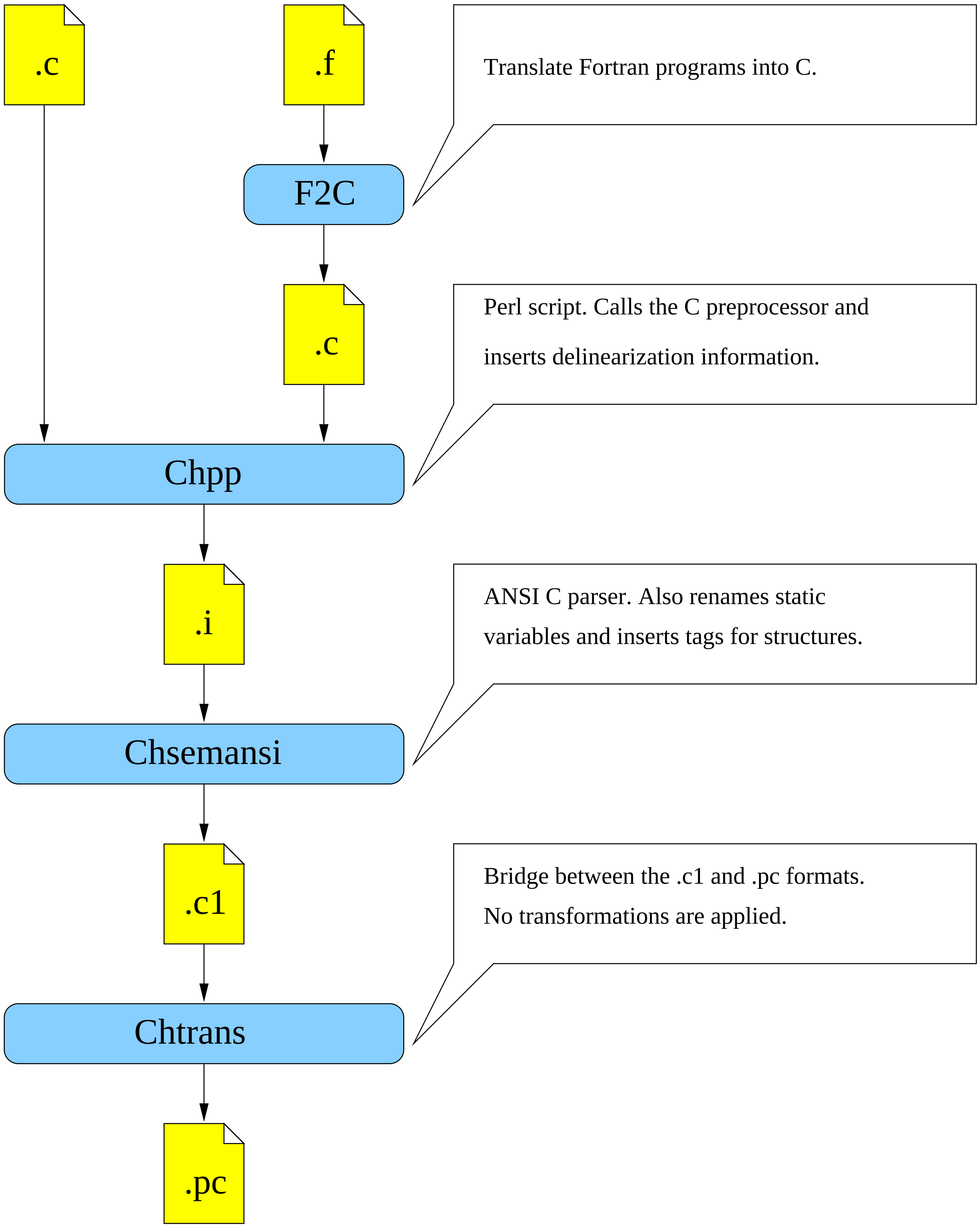


Figure 2.1 Overview of the IMPACT parsing stage.

output of Chsemansi is processed by Chtrans [14] to generate the Pcode IR. Figure 2.1 shows the executables used and intermediate files generated in the parsing stage of the frontend.

2.1.1 Static variables renaming

According to the C language specification [15], an external static variable is only accessible within the file in which it is declared. An internal static variable, in addition to private access by its associated function, remains alive even when the corresponding function exits. The next invocation of that function can retrieve the content of the static variable produced during the previous invocation.

In order to correctly preserve the features of static variables, we first uniquely name each static variable. Suppose a static variable `var` is defined in `filename.c`. After being renamed by Chsemansi the format of its new name is `CHST_filenamei_n_var`, where

- `CHST` is acronym for **CH**semansi renamed **ST**atic variables
- `filenamei` is the respective file where the variable is defined. Originally the filename is filename.c, but after preprocessing the extension name is changed to .i, and with the dot suppressed it finally becomes filenamei. If there are no source files with the same filename, this guarantees the uniqueness of the new name across files.
- `n` is an ascending sequence number incremented whenever a static variable is defined. This guarantees the uniqueness of the new name within the same file.
- `var` is the original name of the variable. This field helps the compiler developers debug their code. The future IMPACT debugger can retrieve the original variable name by this field.

<pre>int i;
static int j;
int foo1()
{
    static int i;
    static int j;
}</pre>
	<pre>int i;
	int CHST_statici_0_j;
	int CHST_statici_1_i;
	int CHST_statici_2_j;
	int foo1 ()
	{
	}</pre>

(a)	(b)

Figure 2.2 Renaming static variables: (a)original code, (b)after renaming.

- `'_'` is used as a separator. Also, all the non-alpha-numerical characters in the original filename are replaced by `'_'`.

Additionally, since internal static variables stay alive as long as the program is alive, Pcode must allocate them in the data segment like global variables, instead of in the activation record. Figure 2.2 shows a simple example called `static.c` of how Chsemansi and Pcode handle static variables. The three static variables in the original code, two inside function `foo1` and one declared outside `foo1`, are renamed and declared globally.

2.1.2 Tag insertion

In many instances structures are used as user-created data types. In `typedef` declarations, the tag name for the structure is often not specified by the programmer. But later in the the compiler all the user-defined types need to be reduced to either generic data types or aggregate data types based on generic types, which requires each aggregate data type have a tag name. Therefore Chsemansi inserts a compiler-generated tag for each tagless structure. The naming convention for an inserted tag is `CH_filename_line_column`, where

```
typedef struct {                    struct CH_structi_1_16 {
    int x;                              int x;
    int y;                              int y;
} coordinate;                       };
                                    struct CH_structi_1_16 p1;
coordinate p1, p2;                  struct CH_structi_1_16 p2;

       (a)                                   (b)
```

Figure 2.3 Tag insertion: (a) original code, (b) after insertion.

- `CH` designates that this name was created by **CH**semansi.
- `filename` is the respective file where the structure is encountered.
- `line` is the line number in the above file where the structure is declared.
- `column` is the column number at the above line where the structure is declared.

The reason for choosing the line number and column number instead of an ascending sequence number is that structures are often declared in the header files and then included by many other source files. We do not want to create redundant copies of the same structure declaration simply because they are included by different files. In that case two objects of the same type could be mistakenly treated as of different types. This tagging approach has been proven to be viable through all the SPEC benchmarks and other test programs. Figure 2.3 shows how the IMPACT frontend handles tagless structures. In this example there are one user-defined type `coordinate` and two instances `p1` and `p2` of that type. Pcode inserts a tag for the `coordinate` structure and uses the tag to declare `p1` and `p2`. The filename is assumed to be struct.c.

2.2 Overview of the Pcode Modules

Pcode is the high-level intermediate representation in the IMPACT compiler. Pcode carries source-level information and performs high-level transformations. The following sections will discuss Pcode IR and activities performed in the Pcode environment.

2.2.1 Pcode intermediate representation

The external file representation of the Pcode IR looks like the LISP [16] language because the basic elements in Pcode are atoms and parentheses. All the constructs, operators, constants and variables in C are assigned corresponding atom names. Internally Pcode rebuilds the C program structures from these lists. As in C, the first level constructors include type definitions, global variable declarations, structure declarations, and function definitions. Within function definitions there are local declaration and statement sections, and expressions are connected within statements.

Pcode statements are represented very closely to their C counterparts. For example, an `if` statement has an expression field pointing to the conditional expression, as well as two statement fields pointing to the true-part and false-part respectively. All the three iterative statements, *for-loops*, *while-loops* and *do-loops*, are also preserved as in C code with the condition expression, optional initialization and iterative expressions, as well as the loop body fields. Later they are flattened and represented as basic blocks connected by `if` statements in the Hcode IR. Unlike statements, expression representations are different than in C code because prefix notation is used instead of infix notation. For example, `"3 + 5"` is represented as `"+ 3 5"`. Prefix representation has the advantage over infix representation of removing the need for parentheses to determine the precedences of operators. It also facilitates the construction of the abstract-syntax tree (AST) [17], by which expressions are linked together as nodes. Most of the transformations performed in Pcode operate on the abstract-syntax tree.

2.2.2 Local variable renaming

In C, any identifier should be unique within one scope, while the same identifier name can be reused in different scopes. A scope could be the global variable scope or a compound statement scope. Pcode observes this by reserving 0 for the global scope and assigning a unique ascending number to each compound statement. So if the same identifier `i` is used as a global variable and local variables in two different compound statements, the local ones will be distinguished by appending the unique scope number after the original name to become `P_i___n` and `P_i___m` respectively, where n and m are the associated scope numbers for the surrounding compound statements.

2.2.3 Semantic Analysis

Chsemansi, the parser of the IMPACT compiler, only performs syntax checking which ensures that the program complies with the language grammar. To ensure that the program components meaningfully fit together, the compiler needs to perform semantic analysis [17]. For example, it makes no sense to dereference a pointer which is the sum of a function pointer and an integer pointer, though it is syntactically correct. Therefore, a lot of effort is put into Pcode to perform semantic analysis.

On the other hand, the compiler should allow certain kinds of type mixture. For example, with type coercions inserted by the compiler, it is legal to add a floating-point number to an integer number, Sometimes a compiler fails to produce correct result because it fails to insert the necessary type coercions, while another time the compiler fails to produce fast code because

```
main()
{
    char c;
    short s;

    c = -1;
    s = c - 255;
    printf("output=%d\n", s);
}
```

(a)

```
main()
{
    char c;
    short s;

    c = -1;
    s = (short)((int)c - 255);
    printf("output=%d\n", s);
}

%output=-256
```

(b)

```
main()
{
    char c;
    short s;

    c = -1;
    s = (short)(c - (char)255);
    printf("output=%d\n", s);
}

%output=0
```

(c)

Figure 2.4 Type coercions in expressions: (a) original code, (b) correct type conversion, (c) incorrect type conversion.

it inserts redundant type coercions. Pcode inserts minimal type casts to achieve the fast and correct result. For example, the code in Figure 2.4(a) subtracts an integer constant from a character variable and assigns the result to a short integer variable. The proper promotions should first cast the character to an integer, then perform the subtraction, and then cast the result to a short integer before assigning it to the destination variable. Figure 2.4(b) shows the wrong procedure, which casts the integer constant to a character first. This yields the wrong answer 0, instead of the correct answer -256.

CHAPTER 3

PREREQUISITES FOR PINLINE

Before invoking Pinline, the source program needs to undergo three preliminary operations, which are statement restructuring, expression flattening and file splitting. Statement restructuring is employed on three kinds of C statements, *expression-statements*, *if-statements* and *loop-statements*, to facilitate expression flattening. After statement restructuring, expression flattening reduces the height of the abstract-syntax trees in order to increase the number of inlinable functions. Finally, file splitting decomposes the source program into one function per file so that callee functions can be conveniently extracted. In this chapter, the general idea of expression flattening will be examined first, followed by the explanation why some statements need to be restructured, then a detailed description about file splitting is given.

3.1 Expression Flattening

Pinline is a high-level source-to-source inliner. Since Pcode is very similar to the original C code, expanding a first-level function call site is trivial. Here "first-level" means an expression-statement which consists of either a single function call or an assignment expression with a function call as the right-hand-side operand. The inlining procedures of first-level call sites are:

- If the function call composes the entire statement, insert expressions that assign the actual parameters to the corresponding formal parameters of the callee function. Then add a label to the end of the callee function and replace every `return` statement in the callee

```
void my_printf(char *str)
{
    printf("%s", str);
}

main()
{
    my_printf("Hello\n");
    return;
}
```

(a)

```
main ()
{
    {
        char *str;

        str = "Hello\n";
        printf("%s",str);
        goto P_my_printf_L___3;
    }

P_my_printf_L___3:
    return;
}
```

(b)

Figure 3.1 Inlining a simple function call: (a) original code, (b) after inlining.

function with a `goto` statement which goes to the newly inserted label. Then replace the call expression with the body of the callee function to finish inlining. Figure 3.1 shows the stated transformation.

- If the call site is on the right-hand side of an assignment expression, the only difference from the transformation of a plain function call is to replace every `return` statement with an assignment expression which assigns the return value to the left-hand-side variable. Figure 3.2 shows this transformation.

However, not every call site can be handled by the above procedures because the C grammar allows a function call to occur at any place wherever an expression is allowed. For example, function calls could be used directly in an expression like "`i = foo1() + foo2()`". In this case to expand the callee functions is not as trivial as in the examples of Figure 3.1 and 3.2 since

```
square(int i)
{
    return i*i;
}

main()
{
    int j, k;

    j = 10;
    k = square(j);
    return;
}
```

(a)

```
main ()
{
    int j;
    int k;

    j = 10;
    {
        int i;

        i = j;
        k = i * i;
        goto P_square_L___3;
    }

P_square_L___3:
    return;
}
```

(b)

Figure 3.2 Inlining a function call in an assignment expression: (a) original code, (b) after inlining.

the C grammar does not allow compound statements to appear at the places where expressions are expected.

In order to inline `foo1()` and `foo2()`, we need to transform the above code into

```
temp1 = foo1();

temp2 = foo2();

i = temp1 + temp2;
```

Now since both `foo1()` and `foo2()` appear as first-level function calls, they can be handled by the inliner very easily. This transformation is called *Expression Flattening* and will be explained in Section 3.1.1.

3.1.1 Flattening methodology

As stated in Section 2.2.1, the Pcode IR is maintained hierarchically with the abstract-syntax tree format. Pcode defines three data structures, `FuncDcl`, `Stmt`, and `Expr`, to represent functions, statements, and expressions, respectively. The AST representation is used in the expression level. Each function has a pointer which points to the first statement in the function body. All statements within the same function are linked via the *predecessor* and *successor* pointers. Similarly, expressions are linked under their parent statements. Each `Expr` object is treated as a node in the AST tree. Fields in `Expr` related to expression flattening are explained as below:

- **opcode**: *Opcode* is an integer field which records the predefined opcode associated with each operator. By this field the operation of a node can be identified.
- **operand and sibling**: Both the *operand* and the *sibling* fields are pointers which point to other expression nodes. For operators without operands, like variable names and scalar numbers, the *operand* field is set as *NULL*. For unary operators, like negation, the *operand* field is set to the only operand of the operator. For operators with more than one operand, the *operand* field of the operator is set to the first operand, while the *sibling* pointer of the first operand is set to the second operand. Similarly, if the operator has a third operand, it is linked to the *sibling* field of the second operand. That is, all the operands of a operator are chained together via their *sibling* pointers.
- **next**: For expressions separated by commas like `"a, b, c"`, they are connected as a chain via the *next* fields of `a` and `b`.

Figure 3.3 shows the AST of expression `a+b, c*d`.

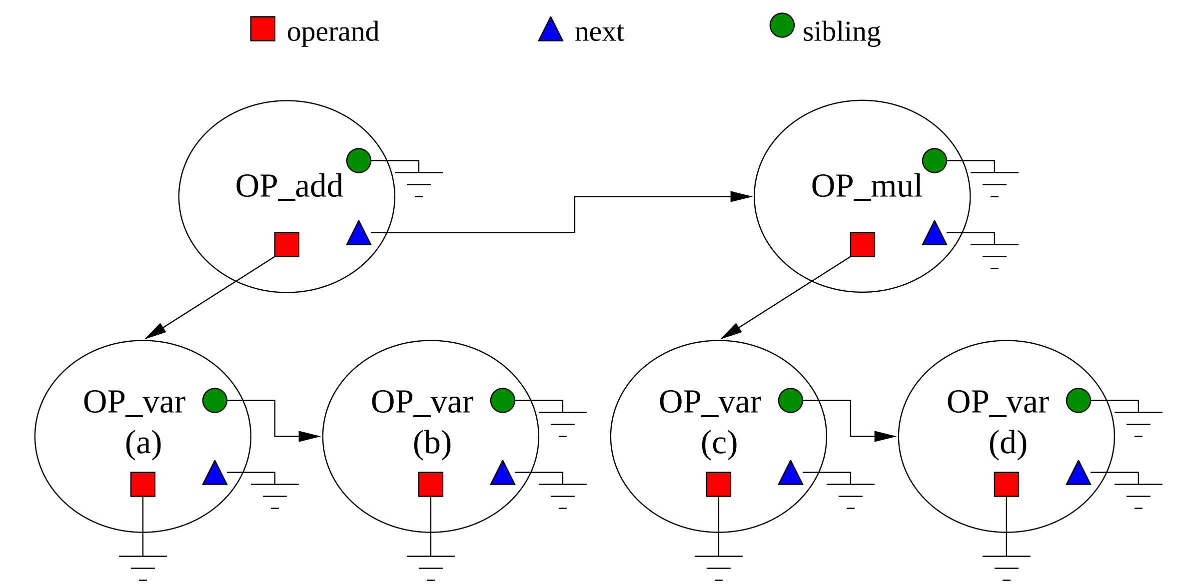


Figure 3.3 Abstract-syntax tree representation of the expression "a+b, c*d".

To flatten an expression, Pcode starts from the first expression in the first statement of a function. For each expression AST, all the internal nodes are visited to see if they need to be marked as *BREAK*. When an internal node is marked as *BREAK*, it means that the subtree rooted at that node will be replaced by a temporary variable to which the evaluation result of the original subtree is assigned. A new statement which evaluates the subexpression and assigns its result to the temporary variable will be created and inserted between the subexpression's parent statement and the predecessor statement of the parent. The algorithm used to analyze the node attributes is presented in Figure 3.4 and the expression `foo1(foo2(), v1), v2++, foo3()` will be used to explain the algorithm.

3.1.1.1 Assign sequence number

In our example, `foo2()` and `v1` are `foo1()`'s operands while `v2++` and `foo3()` are the comma-separated next expressions following `foo1()`. According to the C language specifica-

Mark_Expression_Attribute
input: an abstract-syntax tree
output: the same tree with evaluation attrubutes marked on every node
{

1. Mark explicit attributes.
   - 1.1 If the root node is one of the following operators, `=`, `++`, `--`, `+=`, `-=`, `*=`, `/=`, `%=`, `>>=`, `<<=`, `&=`, `|=`, `^=`, or if it is a function, mark the status of the root node with **F_SIDE_EFFECT**, which is defined as constant 1.
   - 1.2 If the root node is one of the following operators, `&&`, `||`, `?:` (conditional expression), or a function call, mark the status of the root node with both **F_BREAK** and **F_CONTAIN_BREAK**, which are defined as constant 2 and 4, respectively.
   - 1.3 If the root node is a function call, and if it contains arguments, mark each argument node with **F_ARGUMENT**, which is constant 8. The first argument node should appear as the second operand of the root and the rest are connected by their *next* pointers because they are spearated by commas.
2. Recursively call **Mark_Expression_Attribute** on all the *operand* and *next* nodes.

}

Flatten_Expression
input: an abstract-syntax tree with evaluation attributes
output: flattened expression AST with additional statements
{

1. Call **Flatten_Expression** recursively on every *operand* node.
2. If the current node is marked as *BREAK*, create a temporary variable to replace the current node. Also, create a new statement which evaluates the subtree rooted at the current node and assigns its result to the temporary variable. Place the new statement right before the original statement.
3. Call **Flatten_Expression** on the *next* node.

}

Figure 3.4 Flattening algorithm.

tion [15], all the operands of an operator are evaluated first, then the operator itself, followed by the comma-separated next expression. The evaluation order of a function's arguments is not specified and the compiler is free to choose its own implementation. Currently in IMPACT arguments are evaluated from left to right as the appear in the function's declaration. On the other hand, the evaluation order of comma-separated expressions is specified as from left to right. Therefore when the node corresponding to `foo1()` is visited, `1` is assigned to `foo2`, `2` is assigned to `v1`, and `3` is assigned to `v2++`, respectively. In addition, another `1` instead of `4` is assigned to `foo3()` since it is linked as the next expression to `v2++`. The resulting evaluation order of the expression rooted at `foo1()` is

```
foo2() -> v1 -> foo1() -> v2++ -> foo3()
```

3.1.1.2 Mark node attributes

Before marking the current node, **Mark_Expression_Attribute** will be recursively called to mark its *operand* and *next* nodes first. Step 3 of **Mark_Expression_Attribute** will mark `foo1()`, `foo2()` and `foo3()` as *BREAK* since they are function call operators. If we skip step 4 in **Mark_Expression_Attribute** and apply **Flatten_Expression** directly, the following code will be resulted:

```
temp1 = foo2();
temp2 = foo1(temp1, v1);
temp3 = foo3();
temp2, v2++, temp3;
```

However this transformation actually changes the relative evaluation order, because `foo3()` is entered before `v2` is incremented. Assume that `v2` is a global variable whose value is initialized as `0` and `foo3()` references `v2`. To execute the program correctly, `foo3()` should see `1` instead of `0` in `v2`. To guarantee that, step 4 in **Mark_Expression_Attribute** needs to be executed. What it does is simply to propagate $BREAK$ forward along the chain of *operand* and *next* nodes based on their sequence numbers. In this example `v2++` will be marked as $BREAK$ because its next node, `foo3()`, is a function call. The correct transformation is

```
temp1 = foo2();
temp2 = foo1(temp1, v1);
temp3 = v2++;
temp4 = foo3();
temp2, temp3, temp4;
```

3.2 Statements Restructuring

The expression flattening method presented in Section 3.1 only works for straight-line code. That is, it assumes that there is no change of control-flow between adjacent expressions and statements. However, there are several cases in the C-language constructs which contain implicit change of control-flow. These constructs can be divided into two categories. The first category contains boolean expressions and conditional expressions where whether to evaluate an expression or not depends on the result of the proceeding expressions in the same statement. The other category contains loop-constructs where adjacent statements and expressions may be executed different times. The ways to restructure these constructs so that the method described in Section 3.1 becomes applicable are described in the following sections.

3.2.1 Expression-statements

Expression	Original code	Restructured code

Logical-AND expression	`result = expr1 && expr2;`	`result = 0;`
		`if (expr1)`
		`    result = expr2 != 0;`

Logical-OR expression	`result = expr1 || expr2;`	`result = 1;`
		`if (expr1 == 0)`
		`    result = expr2 != 0;`

Conditional-expression	`result = condition ?`
	`    expr1 : expr2;`
		`if (condition)`
		`    result = expr1;`
		`else`
		`    result = expr2;`

Table 3.1 Code restructuring for removing intra-statement control-flow dependences.

In C, constructs like *boolean* expressions and *conditional* expressions have intra-statement control-flow dependences. Table 3.1 shows methods to restructure the code so that these intra-statement dependences are turned into inter-statement dependences.

3.2.1.1 Boolean expressions

For boolean expressions, the dependence exists since expressions are evaluated in the *short-circuit* style [17]. That is, the evaluation stops as soon as the result is known. In such case we cannot blindly pull expressions out of their original locations otherwise the original intra-expression control-flow dependence will be broken.

For a *logical-AND* expression as shown in Table 3.1, `expr1` is evaluated first. If its result is FALSE, `expr2` is not evaluated and FALSE is assigned to result. The restructured code still obeys this rule while the control-flow dependence is now explicit. The situation for a *logical-OR* expression is similar to that of a *logical-AND* expression.

3.2.1.2 Conditional expressions

For a *conditional* expression, depending on the result of `condition`, as shown in Table 3.1, only one of the two possibles destination expressions, `expr1` or `expr2`, will be evaluated. If we directly apply the method presented in Section 3.1, both destination expressions can be executed simultaneously because that method will not enforce the control-flow dependence. The proposed transformation makes the dependence explicit.

3.2.2 If-statements

A simple if-statement with boolean expressions as the predicate actually contains multiple branches because short-circuit evaluation is used for the predicate. These hidden branches need to be transformed into explicit branches so that more accurate control-flow information can be provided to the backend of the compiler to perform optimizations like superblock formation, hyperblock formation and static branch prediction. Figure 3.5 (a) shows a simple if statement and its associated control-flow graph with implicit branches is shown in Figure 3.5 (b).

To expose all the branches embedded in the boolean expressions, the method shown in Section 3.2.1 can be applied here, too. The resulting code is shown in Figure 3.5 (c). However, the potential problem of this transformation is the live range of those temporary variables. For example, `temp1` is initialized with 0 and remains live across the evaluations of `expr1`, `expr2` and `expr3`. If a boolean expression is the conjunction of `n` expressions, `n-1` temporary variables will be generated, which impose great pressure to the register allocator. Therefore a revised transformation is needed.

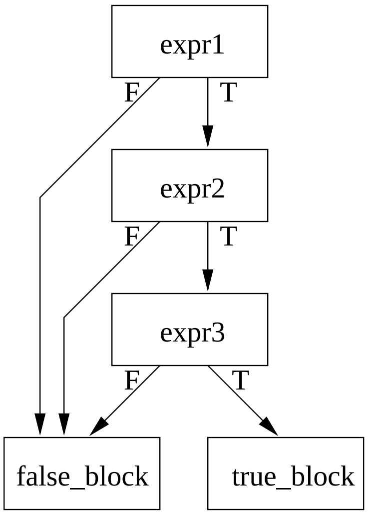
Figure 3.5 (d) shows a better transformation which is from the observation that since the results of expressions in the predicate are only used to change the flow of control, there is

```
if ( expr1 && expr2 && expr3 ) {
    true_block;
}
else {
    false_block;
}
```

(a)

```
temp1 = 0;
temp2 = 0;
if ( expr1 )
    temp2 = expr2 != 0;
if ( temp2 )
    temp1 = expr3 != 0;
if ( temp1 ) {
    true_block;
}
else {
    false_block;
}
```

(c)



(b)

```
if ( expr1 )
    if ( expr2 )
        if ( expr3 )
            goto true_label;
        else
            goto false_label;
    else
        goto false_label;
else
    goto false_label;
{
true_label:
    true_block;
    goto out_label;
}
{
false_label:
    false_block;
}
out_label:;
```

(d)

Figure 3.5 Restructuring if-statements: (a) original code, (b) implicit branches, (c) inefficient restructuring, (d) better restructuring.

no need to create temporary variables to hold temporary values. Consequently, the register pressure is not increased because of restructuring the statement.

3.2.3 Loop-statements

Flattening expressions in loops is also different than flattening other expressions because depending on the context, a single occurrence of an expression in the loop could be executed multiple times. Therefore the position to place the flattened expression is critical. Table 3.2 shows the templates to restructure *for-loops*, *while-loops* and *do-loops*. Each of them will be explained in the following sections.

3.2.3.1 For-loops

For the for-loop example shown in Table 3.2, if `initial_exprs`, `conditional_exprs` and `iterative_exprs` contain function calls, we cannot perform inline expansions within the parentheses because compound statements are not allowed there. If we do not restructure the for-loop but directly apply the flattening algorithm stated in Section 3.1, these expressions will be evaluated outside the loop and done so only once. To execute the loop correctly, the `conditional_exprs` should be executed right before entering the first iteration to initialize the iterative conditions. The `conditional_exprs` must be executed once at the beginning of every iteration because they are executed to determine if the loop body will be entered or not. Similarly, the `iterative_exprs` should be executed once at the end of every iteration to update the iterative conditions.

In order to obey the for-loop semantics while enable the possibility of function inlining, the `initial_exprs` are moved right outside of the for-loop, the `conditional_exprs` are moved into the very beginning of the loop body as the predicate of an if-statement which can terminate

<table>
<tr><th>Loop</th><th>Original code</th><th>Restructured code</th></tr>
<tr><td>For-loop</td><td><pre>for (initial_exprs;
     conditionl_exprs;
     iterative_exprs)
{
        :
    if (...) continue;
        :
}</pre></td><td><pre>initial_exprs;
for (;;)
{
    if (!conditional_exprs) break;
        :
    if (...) goto new_continue;
        :
new_continue:
    iterative_exprs;
}</pre></td></tr>
<tr><td>While-loop</td><td><pre>while (conditional_exprs)
{
    :
}</pre></td><td><pre>while (1)
{
    if (!conditional_exprs) break;
        :
}</pre></td></tr>
<tr><td>Do-loop</td><td><pre>do
{
        :
    if (...) continue;
        :
} while (conditional_exprs);</pre></td><td><pre>do
{
        :
    if (...) goto new_continue;
        :
new_continue:
    if (!conditional_expr) break;
} while (1)</pre></td></tr>
</table>

Table 3.2 Code restructuring for loop-statements.

the iteration of the for-loop, and the `iterative_exprs` are moved into the very bottom of the loop body. If there are `continue` statements in the loop body, they need to be changed into `goto` statements otherwise the `iterative_exprs` will not get executed because now they are not in the for-loop parentheses. To do this we need to attach a label to the `iterative_exprs` so that they are directly accessible.

3.2.3.2 While-loops

A while-loop is easier to restructure than a for-loop since it contains the `iterative_exprs` only. We perform similar transformation by moving the `conditional_expressions` into the beginning of the loop body and leaving a "1" in the original place.

3.2.3.3 Do-loops

A do-loop is different from a while-loop in that the `conditional_exprs` are evaluated at the end of each iteration. To restructure a do-loop, we move the `conditional_exprs` to the end of loop body as the predicate of an if-statement which can terminate the loop. The original place of the `conditional_exprs` is replaced by a "1". Also, all the `continue` statements are transformed into `goto` statements otherwise the `conditional_exprs` are not reached. We also need to attach a new label to the if-statement.

3.2.4 Expression flattening results

Table 3.3 shows the impact of code flattening on the number of expandable call sites for the SPEC INT benchmarks. We can see that without code flattening, only 59.3% of the total function calls can be inlined at the Pcode level, which does not satisfy the need to achieve high ILP. After flattening, all the function calls can be inlined at the Pcode level.

3.3 File Splitting

For C programs, a single file usually contains more than one function. Because Pinline is a whole-program inliner, it is possible that we will want to inline function `fooA1` in `fileA.c` into function `fooB1` in `fileB.c`. In this case the compiler must read `fileB.c`, locate function

Benchmarks	# of expandable call sites	# of total call sites	percent
008.espresso	1706	2693	63.3%
022.li	923	1271	72.6%
023.eqntott	271	369	73.4%
026.compress	83	130	63.8%
072.sc	788	1458	54.0%
085.cc1	5067	8415	60.2%
099.go	1683	2085	80.7%
124.m88ksim	980	1499	65.4%
126.gcc	11701	19820	59.0%
129.compress	53	64	82.8%
130.li	923	1271	72.6%
132.ijpeg	884	1654	53.4%
134.perl	2326	4372	53.2%
147.vortex	4411	8536	51.7%
mean			64.7%

Table 3.3 Distribution of expandable and total call sites without code flattening.

`fooB1`, locate the function call in `fooB1` to `fooA1`, read `fileA.c`, locate function `fooA1`, perform inline expansion, then write back `fooB.c`. Additionally, the most difficult task is to include all variables and types referenced by `fooA1` in `fileA.c` into `fileB.c`. This is doable, but very complicated.

To address the above difficulty, we split the program into multiple files with a tool called Psplit. After splitting, each file contains only one function. If all the static variables and duplicated structure tags are properly renamed, they can be exposed to all the functions within the program, even though originally they are not from the same file. Section 3.3.1 presents the general idea of Psplit and Section 3.3.2 investigates the method of renaming structures.

3.3.1 Overview of program splitting

In the C language, the names of global variables are required to be unique within the program. Also, the names of static variables are made unique by Chsemansi. If structure names are also unique, we can safely make all the global and static variables visible to all the functions. That is, we split the program by gathering all variable definitions into a file called *data.pcs*, all variable declarations into a file called *extern.pch*, all structure declarations into a file called *struct.pch*. Then we decompose the program into one function per file, where each file contains no global data definitions. All the global data information is made visible to the functions by including *extern.pch*. Figure 3.6 gives a clear look at the dependences between these files. Codes will be generated for files ending with *.pcs*. Files ending with *.pch* are header files and therefore no code will be generated for them.

3.3.2 Structure renaming

The C language grammar only requires structure names to be unique within a single file. Therefore, defining structures with the same tag name but different fields is allowed across files. For example, one can define a structure called `S` in `fileA.c` to have two integers and redefine `S` to have three integers in `fileB.c`. However, this introduces some problems into the program splitting algorithm.

Psplit renames structures by maintaining a table which keeps track of all defined structures. Whenever a new structure is encountered, its name is checked against the structures in the table. If a structure with the same tag name is found in the table, their fields are checked. If any difference is found, a new tag name is created for the new structure and all references in the current file to that structure will be redirected to use the new name.

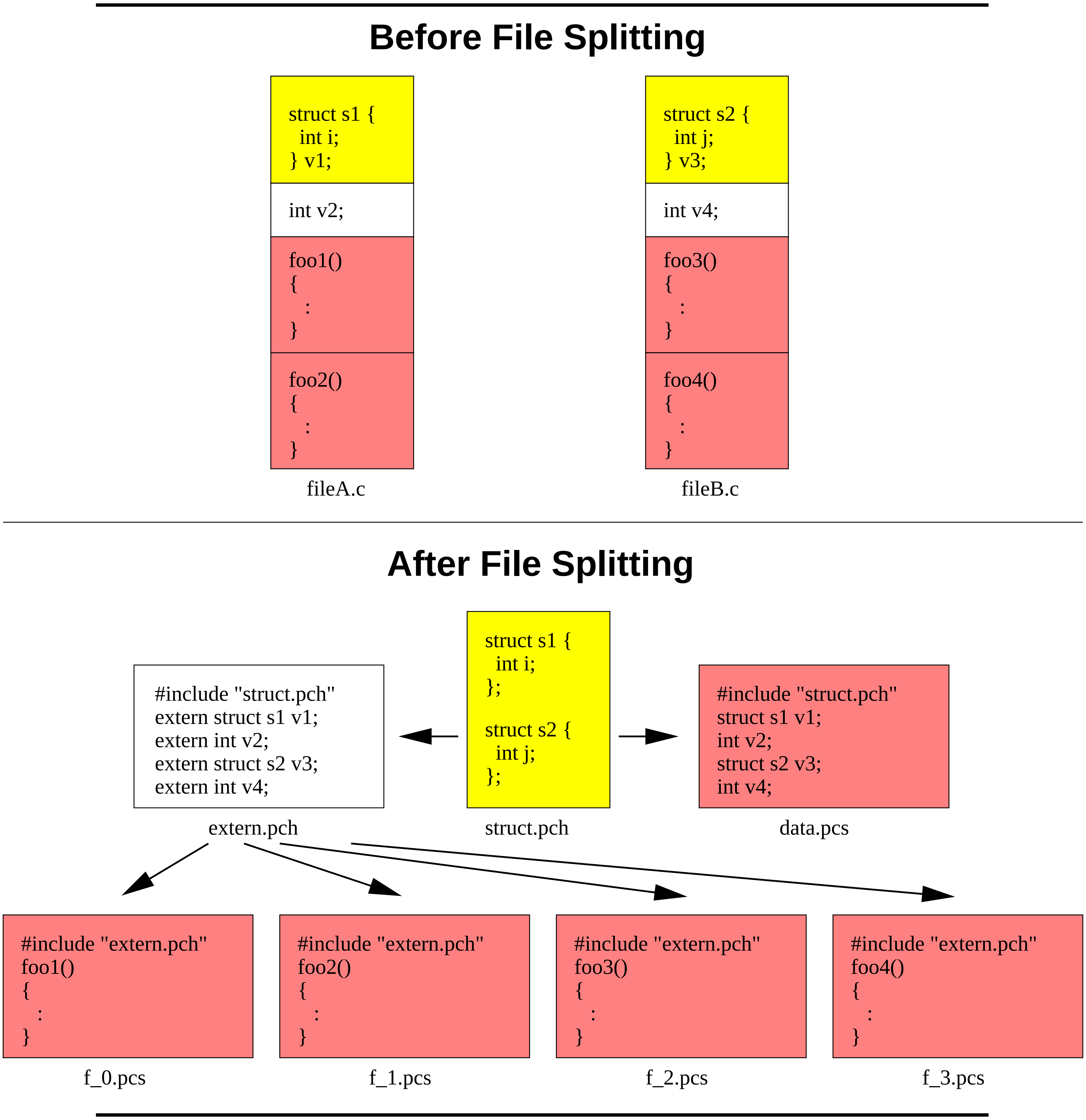


Figure 3.6 Overview of program splitting.

CHAPTER 4

IMPLEMENTATION OF PINLINE

Pinline is a profile-driven automatic source-to-source inliner. It takes annotated Pcode files with profile weights then performs function inlining. According to Table 3.3, the program 126.gcc contains as many as nearly 20,000 function call sites. It is impossible to inline all of them because the final code size expansion would be catastrophic. To benefit from function inlining, we would like to inline frequently executed function calls as well as callees with small body size because inlining a rarely called function will not improve performance too much, while inlining a huge function could potentially pollute the I-cache. Section 4.1 explains how Pinline works step-by-step at a high level. Meanings of parameters used to tune the outcome of Pinline are explained in Section 4.2. Finally, an example is given in Section 4.3 to illustrate a variety of function inlining patterns.

4.1 Overview

Pinline assumes that the source program has been flattened, profiled, and split. Also, the program has been analyzed by PIP, which represents for *Pcode InterProcedural analyzer* [11]. This tool resolves function pointers and attaches a list of possible callee names to each function-pointer call site. Figure 4.1 shows the steps that the program should have gone through before it reaches Pinline. The desired consequence is that Pinline simply needs to inline first-level function calls. Pinline processes the program in multiple steps as stated below:

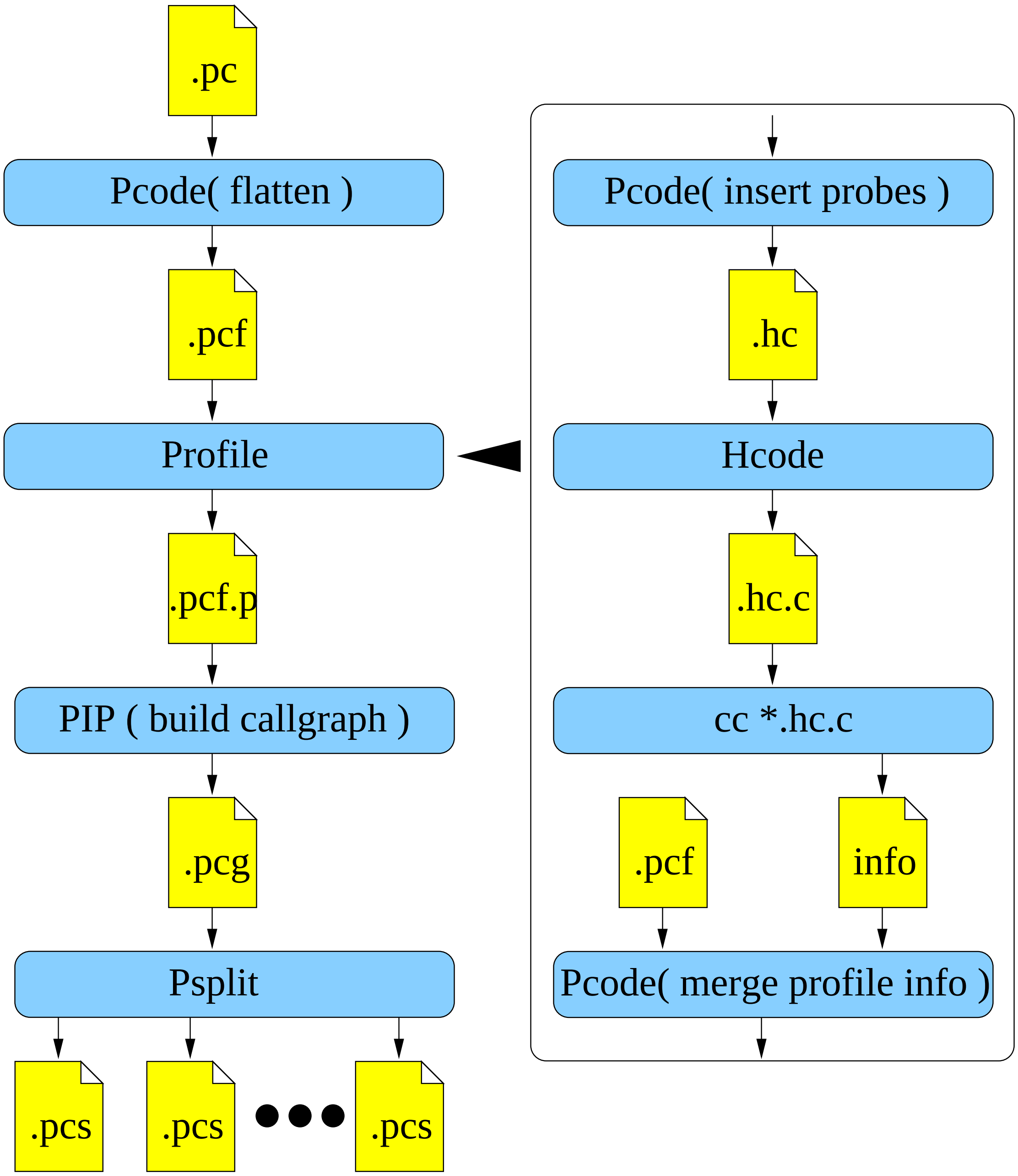


Figure 4.1 Overview of prerequisites for Pinline.

(1) **Gather function-file mapping information**. Since the source program has gone through Psplit, Pinline needs information to locate functions from the split files. The format of the mapping information is:

```
(map 1 (fileA.pc foo1) to (f_0.pcs foo1))
```

where `map` is the keyword, `1` means that the function can be inlined, `fileA.pc` is the original file which contains the function `foo1`, and `f_0.pcs` is the new file which contains the same function `foo1`.

(2) **Mark function calls**. Because the program has been profiled, all function calls have associated weights. Pinline searches the whole program for important function calls. These function calls must meet all of the following criteria:

- If the callee function contains more than 3 operations, the execution frequency of the call site needs to be higher than `min_expansion_weight`, a parameter of Pinline which will be explained in the next section. For small functions, as long as they are ever called, they are eligible for being inlined.
- The body of the callee must be available in Pcode format. This means that Pinline cannot inline library function calls.
- The callee function must not use a variable-length argument list.

Those eligible function call sites will each be assigned a unique ID number for later identification. Together with the ID number, the call site's caller and callee names and its weight are grouped and inserted into a priority heap. Their usage will be clear after the example shown in Section 4.3. In order to favor small and frequently called functions, the key of the priority heap is defined as: $\frac{W}{\sqrt{S}}$, where W is the call site weight and S is the size of the callee's body.

(3) **Inline functions**. Pinline continues to extract the highest priority function call out of the heap and tries to inline the callee until the heap is empty. If the body size or stack size of the resulting function after expansion exceeds either one of the two parameters, `max_function_size` and `max_sf_size_limit`, the callee function will not get inlined.

4.2 Parameters

Pinline is controlled by a set of parameters. Important ones are explained as follows:

- **Print_inline_stats -** setting this parameter to yes enables Pinline to generate a log file which records information pertaining to each function including the body and stack size, the frequency that the function is called. Also, the file keeps track of every decision Pinline made about to or not to inline a certain function.
- **max_sf_size_limit -** parameter specifies the maximum stack frame size a function can reach after inlining. The size is measured by the number of bytes allocated in the function.
- **max_expansion_ratio -** parameter controls the maximum ratio of program size expansion after inlining.
- **max_function_size -** parameter controls the upper bound on the size that one function can reach. The size is measured by the number of operations contained in the function.
- **min_expansion_weight -** parameter specifies the lower bound on function call site frequencies. Function calls with lower weight than this will not be expanded.

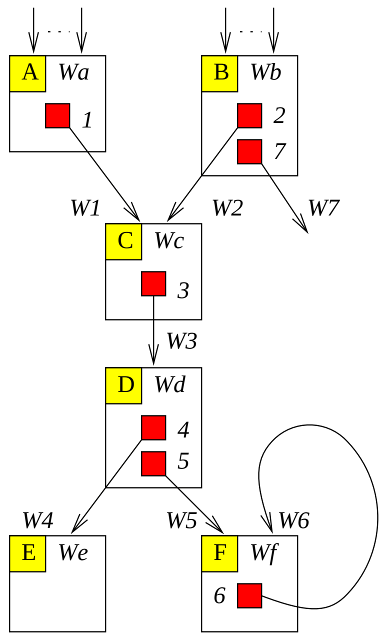


Figure 4.2 Sample call graph.

- **inline_function_pointers -** setting this parameter to yes enables Pinline to expand function pointer call sites. The detail will be explained in Section 4.3.3.

4.3 Call site Samples

The most important task for Pinline is to make intelligent and correct inline decisions. As stated before, these decisions are based on the call site frequency and the callee size. Therefore Pinline must ensure that the frequencies are correctly distributed and the sizes are correctly accumulated during inlining. Figure 4.2 shows a sample call graph with five kinds of function calls. All five cases will be explained later.

In this call graph, there are six functions A, B, C, D, E, F with first instruction execution frequencies as *Wa, Wb, Wc, Wd, We, Wf*, respectively. In addition, there are seven call sites *1, 2, 3, 4, 5, 6, 7* with frequencies *W1, W2, W3, W4, W5, W6, W7*. Since it is a graph, functions can be referred to as nodes and call sites can be referred to as arcs. The following formulas hold for these nodes and arcs:

$$
\begin{aligned}
W_a &\geq \sum weights\ of\ incoming\ arcs\ to\ A \\
W_b &\geq \sum weights\ of\ incoming\ arcs\ to\ B \\
W_c &\geq W_1 + W_2 \\
W_d &\geq W_3 \\
W_e &\geq W_4 \\
W_f &\geq W_5 + W_6
\end{aligned}
$$

Notice that the relation is "$\geq$" instead "$=$". It is because the existence of the function pointer arc 7 since the target of arc 7 can not be determined by the compiler. In real programs the relation for most of the nodes is "$=$".

4.3.1 Normal functions

In this subsection, the mechanism for expanding non-recursive and non-pointer functions are examined. The general rule for splitting weights over normal function calls is that if an arc A with weight W_A to node N with weight W_N is expanded, $\frac{W_A}{W_N}$ of N is moved to the caller. That is, the weights of all the new instructions brought to the caller are $\frac{W_A}{W_N}$ out of their original weights and the weights of all the instructions in N become $1 - \frac{W_A}{W_N}$ out of their original weights. And the caller's new body size is the sum of its old size plus the body size of the callee. So is the stack size calculated.

In the sample call graph, arc 4 is the simplest case because E is a leaf node and arc 4 is the only incoming arc to E. After inlining, the function call to E in D is replaced by the body of

E and node E's remaining non-inlined portion is zero if E has never been entered from arc 7. Figure 4.3 (a) shows the resulting call graph.

Inlining D into C via arc 3 totally absorbs D since C is the only caller to D. The difference between arc 3 and 4 is that now call sites 4 and 5 are moved into C, because C now contains the body of D. Figure 4.3 (b) shows the call graph after inlining arc 3. It is also assumed that C has never been called via arc7, otherwise node D would remain in the call graph with a node weight but without incoming arcs.

For arc 1 and 2, since C is the callee of more than one call site, inlining either arc 1 or arc 2 only cannot totally absorb C. Figure 4.3 (c) shows the distributions of C when only arc 1 is inlined. Notice that the weight of node C now becomes $\frac{W_2}{W_c}W_c$, which is W_2. The weight of the new call to D in A is $\frac{W_1}{W_c}W_3$, and the weight of the original call to D in C becomes $\frac{W_c - W_1}{W_c}W_3$. Figure 4.3 (d) shows the case after both arcs 1 and 2 are inlined, assuming arc 7 does not reach node C.

4.3.2 Recursive functions

In this example node F is a self-recursive function which is invoked via D. When arc 5 is inlined, there is no way to totally absorb F because a new arc to F is introduced in D. The original weight of F is distributed such that W_5 is moved into D and $W_f - W_6$ is left in F. Also, the weight of the new arc 6' in D is $\frac{W_5}{W_f}W_6$ and the new weight of the old arc 6 is $\frac{W_f - W_5}{W_f}W_6$. Figure 4.4 shows the resulting call graph.

Inlining a self-recursive function call like arc 6 is a total different story because the caller and the callee are the same function. Inlining this arc cannot totally eliminate the function call but it will reduce function call overheads and increase basic block sizes. The distribution

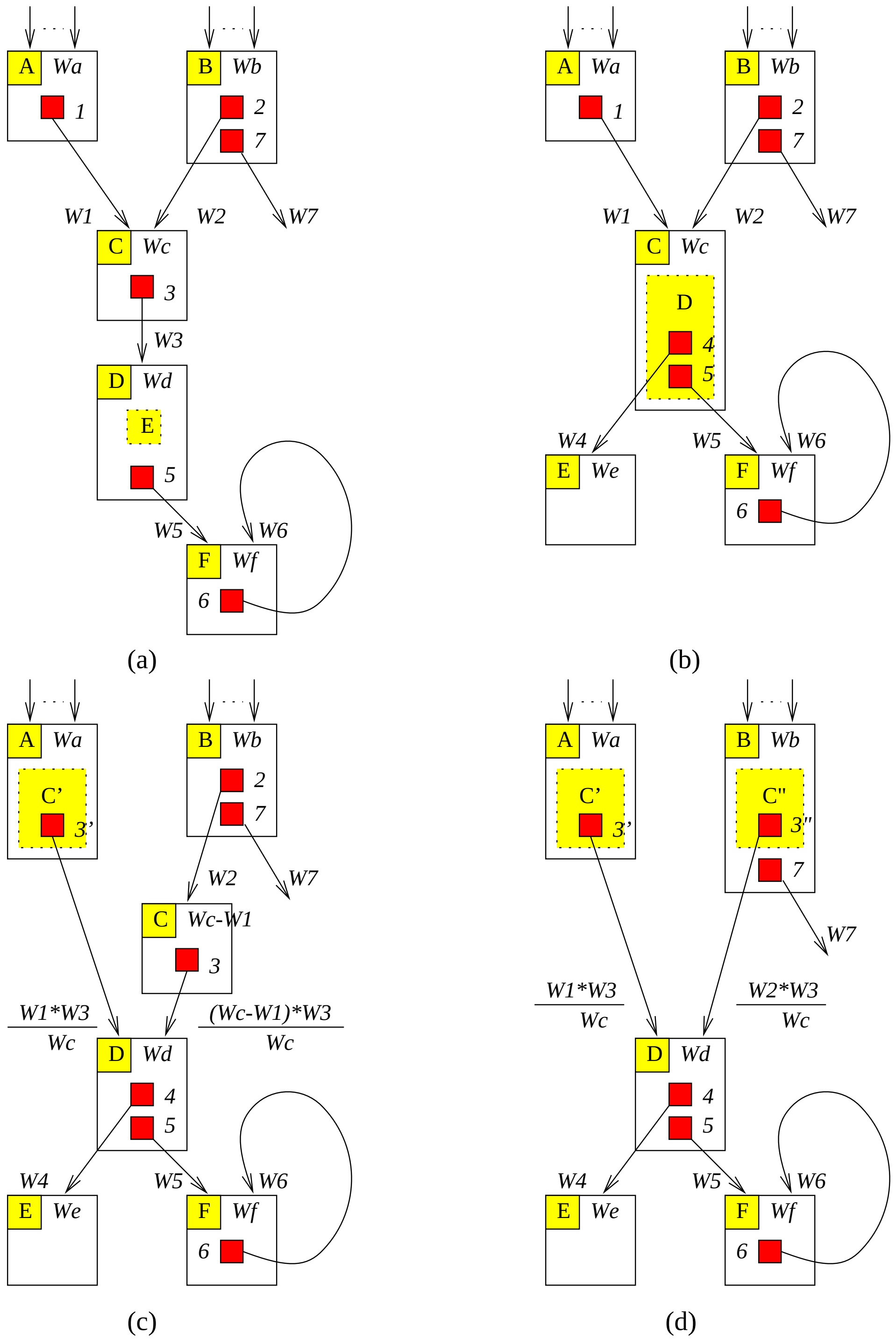


Figure 4.3 Call graphs after: (a) inlining arc 4, (b) inlining arc 3, (c) inlining arc 1, (d) inlining both arc 1 and 2.

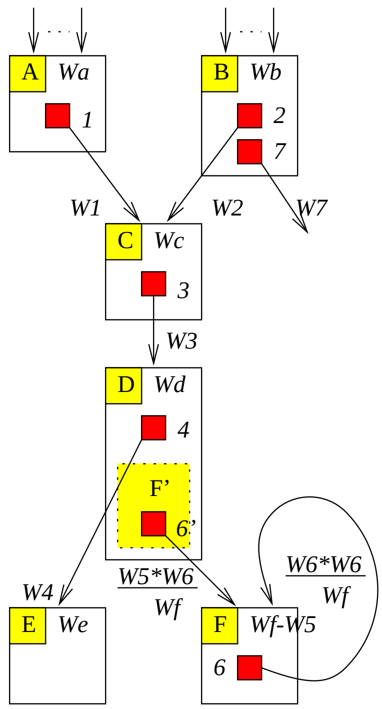


Figure 4.4 Call graph after inlining arc 5.

of weight when inlining self-recursive functions is very different than inlining normal functions in that the caller and callee share the same copy of the body. Unlike inlining normal functions where a portion of the overall weight is dealt out to the caller, the distribution of weight is rearranged within the expanded function.

If the first instruction in F is I1, the weight of I1 should be W_f. After inlining, there are two instances of I1, where I1' is treated as the I1 in the caller while I1" is treated as the I1 from the expanded callee. The following equations should hold for these I1's:

$$\begin{aligned} W_f &= W_{I1'} + W_{I1"} \\ W_{I1"} &= \frac{W_6}{W_f} W_{I1'} \\ W_{6'} &= \frac{W_6}{W_f} W_{I1"} \end{aligned}$$

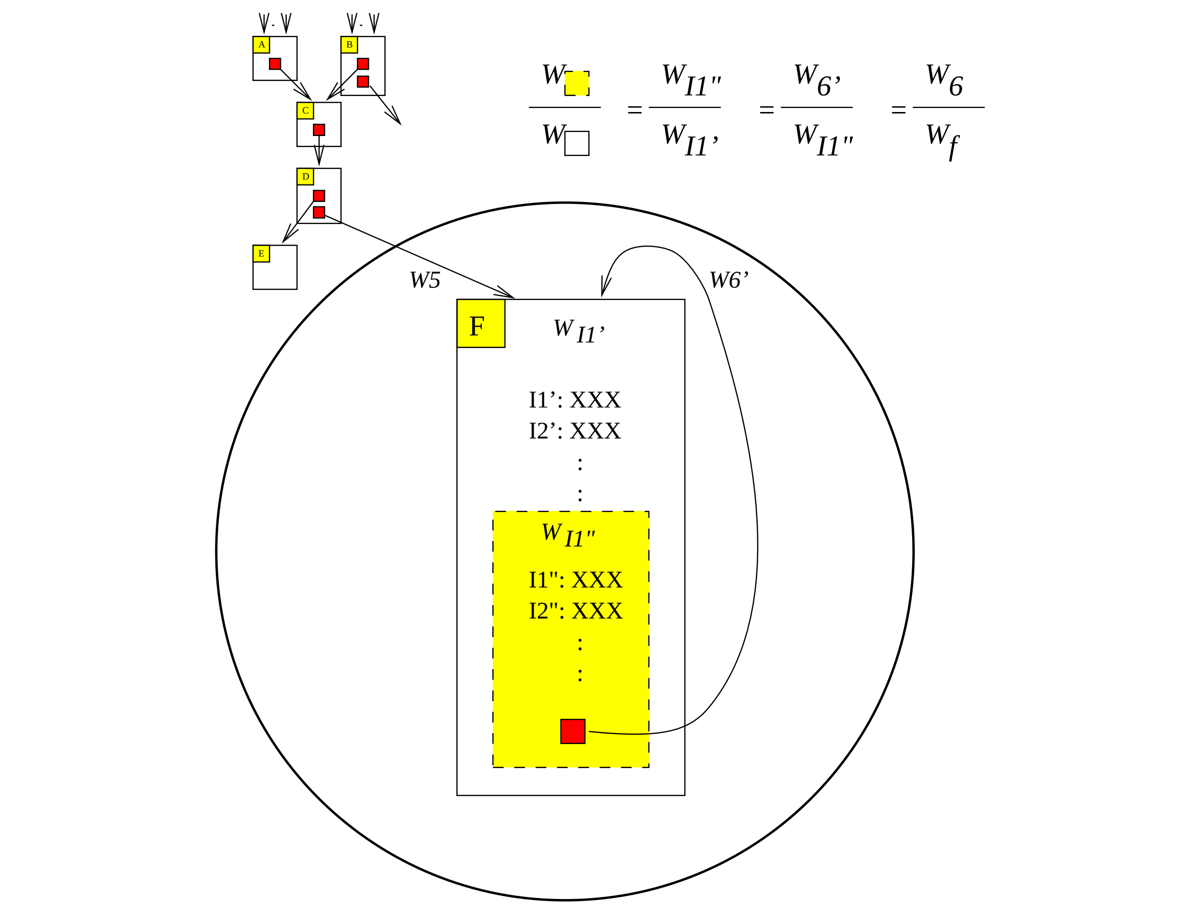


Figure 4.5 Call graph after inlining arc 6.

By solving the equations we get

$$W_{I1'} = \frac{W_f^2}{W_f + W_6}$$
$$W_{6'} = \frac{W_6^2}{W_f + W_6}$$

Figure 4.5 illustrates the distribution of weights in expanded self-recursive functions.

4.3.3 Function pointers

While the existence of function pointers adds flexibility to the programmers, it also adds load the programmers of compilers. That is because the actual callee of a function pointer is not known for sure until run-time. With the help of Pcode interprocedural analysis [11], a list of pragmas for possible callee names is appended to the function pointer call. Then Pinline sorts the list by putting the most frequently called node first, excluding callees with different numbers of parameters. Also, a node whose first-instruction weight equal to the sum of all incoming arcs' weights is not considered as a possible candidate. Then if the pointer is finally resolved to be only one possible name, say, `foo1`, the function pointer call `(*fp_call)()` is transformed into

```
if (fp_call == foo1) {
    foo1();
}
else
    (*fp_call)();
```

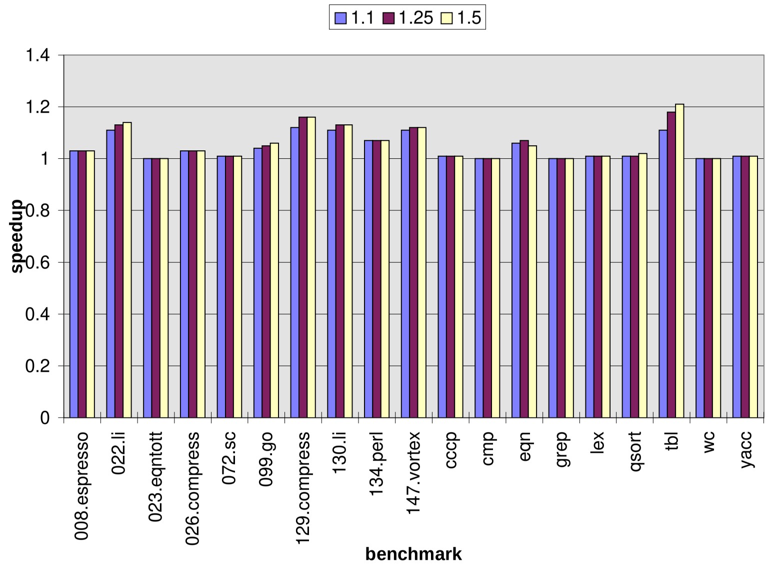
and depending on the size of `foo1()` it could further be inlined. Notice that the original function pointer call is put as the `else` part of the `if` statement as a safe guard. The original weight of the call is evenly spread over the `if` statement. For example, if the original weight of the call is W, then the `if` statement itself will have weight W, the true-part and the false-part will have weight $\frac{W}{2}$. However, after `foo1()` gets inlined into the true-part, $\frac{W}{2}$ will not be deducted from the weight of node `foo1()` because we are not sure if `foo1()` is actually the callee. Leaving more weight with a node does not compromise the priority heap, but negative weight does.

CHAPTER 5

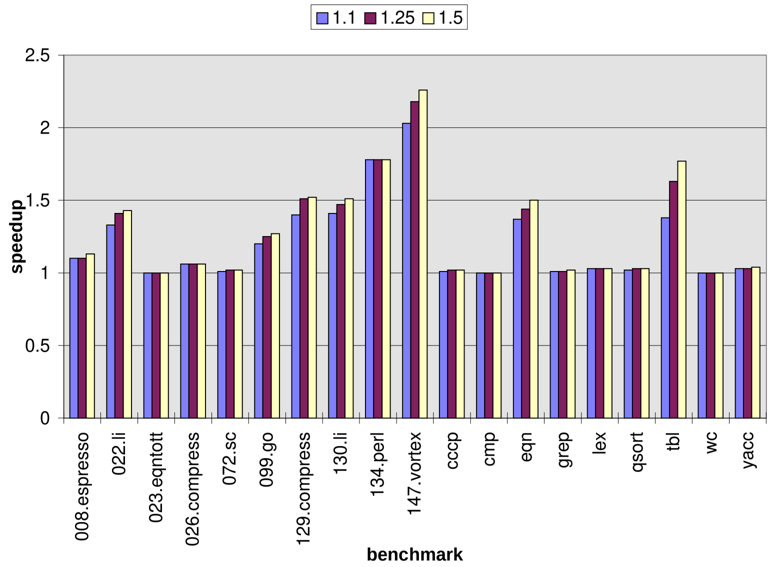
EXPERIMENTAL RESULTS

In this chapter, experimental results from applying Pinline to a suite of integer benchmarks consisting of SPEC92, SPEC95 and common Unix benchmarks are studied. The speedup resulted from performing function inlining on floating-point benchmarks is found to be negligible because those benchmarks are computation-intensive and functions calls rarely occur in the loops. Therefore floating-point benchmarks will not be discussed here. In the experiments, a fully-uniform 8-issue processor with one branch unit is assumed. Based on this configuration, the impact from basicblock and superblock scheduling on three different amounts of code expansion ratios, 1.1, 1.25, and 1.5, are studied. The code size is measured by the number of operations in the Pcode IR.

With basicblock scheduling, the speedup after function inlining, shown in Figure 5.1 (a), mainly comes optimizations made possible after the removal of fences between the caller and callee. Without these barriers more optimization opportunities for local optimizations are created. Figure 5.2 shows a simple example. Originally, function `switch_foo1()` contains a `switch` statement which prints out a string based on the actual parameter. After function inlining, as shown in Figure 5.2(b), the value 20 can be propagated into the switch statement and local optimization can get rid of the switch statement and generate succinct code as in Figure 5.2(c). Besides the increased opportunities of traditional optimizations, the speedup also results from the enlarged scope of the register allocator.



(a)



(b)

Figure 5.1 Speedup on expanded code with: (a)basicblock scheduling, (b)superblock scheduling.

```c
switch_foo1(int i)
{
    switch (i) {
        case 10 :
            printf("10\n");
            break;
        case 20 :
            printf("20\n");
            break;
        default :
            printf("others\n");
            break;
    }
}

main()
{
    switch_foo1(20);
}
```

(a)

```c
main()
{
    int i;

    i = 20;
    switch (i) {
        case 10 :
            printf("10\n");
            break;
        case 20 :
            printf("20\n");
            break;
        default :
            printf("others\n");
            break;
    }
}
```

(b)

```c
main()
{
    printf("20\n");
}
```

(c)

Figure 5.2 Optimizations made possible by function inlining: (a) original code, (b) code after inline expansion, (c) code after inline expansion and local optimization.

Figure 5.1 (b) shows the performance improvement with superblock scheduling techniques. If we compare Figure 5.1 (a) and Figure 5.1 (b), we can find that for those benchmarks with obvious performance improvement in the basicblock case, the speedup is further boosted if superblock scheduling is applied. This is resulted from the enriched freedom of the scheduler since the expanded code form multiple callees can coalesce into a superblock therefore more independent instructions can be intermixed.

As for the amount of inlining, experimental results show that it is highly dependent on the properties of individual benchmarks. Therefore one can use the provided parameters to tune Pinline to get the desired results.

CHAPTER 6

CONCLUSIONS AND FUTURE WORK

This thesis has described the necessary steps to implement a source-to-source automatic inliner. Function call is a very important construct in the modern programming languages. It helps the programmer to write modularized code. However, it imposes great challenge to the compiler because a fence is placed between the caller and callee therefore important optimization and scheduling opportunities are prevented. Given a piece of program, after performing statement restructuring and expression flattening, all the function calls can be exposed to the inliner for code expansion. If a function pointer is resolved by PIP, Pinline can further inline the possible callee.

After code expansion, aggressive ILP optimizations can produce significant speedup on some benchmarks. However, for most of them more potential optimization and scheduling opportunities are still restricted by ambiguous memory dependences. Since the time and space complexity of the the current memory disambiguation algorithm employed by IMPACT is of $O(n)$, where n is the number of memory access, the feasibility of this algorithm on expanded code is limited. Therefore future work is needed to reduce the space and time requirements of the algorithm so that it can be applicable on expanded code.

REFERENCES

[1] P. P. Chang, S. A. Mahlke, W. Y. Chen, N. J. Warter, and W. W. Hwu, "IMPACT: An architectural framework for multiple-instruction-issue processors," in *Proceedings of the 18th International Symposium on Computer Architecture*, pp. 266–275, May 1991.

[2] N. J. Warter and G. E. Haab, "Pcode manual." IMPACT compiler documentation, Center for Reliable and High-performance Computing, University of Illinois, Urbana, IL, 1991.

[3] P. P. Chang, "The Hcode language and its environment." IMPACT compiler documentation, Center for Reliable and High-performance Computing, University of Illinois, Urbana, IL, 1989.

[4] P. P. Chang and W. W. Hwu, "The Lcode language and its environment." IMPACT compiler documentation, Center for Reliable and High-performance Computing, University of Illinois, Urbana, IL, 1989.

[5] W. W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery, "The Superblock: An effective technique for VLIW and superscalar compilation," *The Journal of Supercomputing*, vol. 7, pp. 229–248, January 1993.

[6] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann, "Effective compiler support for predicated execution using the hyperblock," in *Proceedings of the 25th International Symposium on Microarchitecture*, pp. 45–54, December 1992.

[7] R. A. Bringmann, "Template for code generation development using the IMPACT-I C compiler," Master's thesis, Department of Computer Science, University of Illinois, Urbana, IL, 1992.

[8] W. Y. Chen, "An optimizing compiler code generator: A platform for RISC performance analysis," Master's thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1991.

[9] R. G. Ouellette, "Compiler support for SPARC architecture processors," Master's thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1994.

[10] P. P. Chang, S. A. Mahlke, W. Y. Chen, and W. W. Hwu, "Profile-guided automatic inline expansion for C programs," *Software Practice and Experience*, vol. 22, pp. 349–370, May 1992.

[11] D. M. Gallagher, *Memory Disambiguation to Facilitate Instruction-Level Parallelism Compilation.* PhD thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1995.

[12] S. I. Feldman, D. M. Gray, M. W. Maimore, and N. L. Schryer, "A Fortran-to-C converter," Computing Science Tech. Rep. 149, AT&T Bell Laboratories, Murray Hill, NJ, June 1990.

[13] G. E. Haab, "Data dependence analysis for Fortran programs in the IMPACT compiler," Master's thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1995.

[14] N. J. Warter, P. P. Chang, S. Anik, G. E. Haab, S. A. Mahlke, W. Y. Chen, and W. W. Hwu, "Charlie C: A reference manual." IMPACT compiler documentation, Center for Reliable and High-performance Computing, University of Illinois, Urbana, IL, 1991.

[15] B. W. Kernighan and D. M. Ritchie, *The C programming language*. Englewood Cliffs, NJ: Prentice Hall, 2nd ed., 1988.

[16] P. Winston and B. Horn, *LISP*. Reading, MA: Addison-Wesley, 1989.

[17] A. Aho, R. Sethi, and J. Ullman, *Compilers: Principles, Techniques, and Tools*. Reading, MA: Addison-Wesley, 1986.