STATIC PROGRAM ANALYSIS TO ENHANCE PROFILE INDEPENDENCE IN INSTRUCTION-LEVEL PARALLELISM COMPILATION

BY

BRIAN LEE DEITRICH

B.S., Rose-Hulman Institute of Technology, 1988
M.S., National Technological University, 1994

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Electrical Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1998

Urbana, Illinois

STATIC PROGRAM ANALYSIS TO ENHANCE PROFILE INDEPENDENCE IN INSTRUCTION-LEVEL PARALLELISM COMPILATION

Brian Lee Deitrich, Ph.D.
Department of Electrical and Computer Engineering
University of Illinois at Urbana-Champaign, 1998
Wen-mei W. Hwu, Advisor

An instruction-level parallelism (ILP) compiler uses aggressive optimizations to reduce a program's running time. These optimizations have been shown to be effective when profile information is available. Unfortunately, it is not always possible for users to profile their programs. In addition, even if profiling is performed, two other problems can work against the compiler. First, the conditional branch behavior seen for real executions may differ from the behavior seen during profiling; second, important code sections for real executions may be unexercised during profiling.

The objective of this dissertation is to provide a groundwork for an ILP compiler to effectively deal with all three of these issues through the use of static program analysis. For the case when no profiling is performed, this dissertation improves the state of the art in static branch prediction, and investigates the problems associated with static loop-trip-count prediction and static frequency generation. For the case of differing branch behavior, this dissertation proposes the use of the speculative hedge heuristic during acyclic scheduling. Speculative hedge minimizes its dependence on profile information through the use of static analysis, and similar techniques should be developed for other compiler heuristics. Finally, for the case when important code sections are unexercised, this dissertation proposes the use of loop grouping. Loop grouping is a new technique that identifies loops that iterate with the same loop control in multiple locations of the source code. These groups are used to statically predict loop behavior for loops that are untouched during program profiling. In this way, the compiler can

extract information obtained during profiling and apply it to unexercised code. This allows much stronger predictions to be made about unexercised loops, potentially leading to better performance.

DEDICATION

To my wife, Joanna

ACKNOWLEDGMENTS

First, I would like to thank my advisor, Professor Wen-mei Hwu, for his guidance throughout my graduate studies. He provided the necessary intellectual, financial, and professional support that I needed to succeed in graduate school. I look forward to applying the knowledge that I gained in my future career.

Next, I would like to thank the members of my dissertation committee — Dr. Scott Mahlke, Professor William Sanders, and Professor Benjamin Wah. Scott Mahlke has always been a source of good ideas, and he was always willing to spend time with me to discuss the status of my work. William Sanders was very busy, but still found time to spend with me while I was working through some of the issues in my research.

This research would not have been possible without the support of the IMPACT research group — past and present. Members of the group were always willing to spend time to discuss ideas, debate solutions, practice talks, and develop software. I feel very fortunate to have had the opportunity to work in this environment. Roger Bringmann, John Gyllenhaal, Rick Hank, and Scott Mahlke were always willing to help me while I learned how to use the IMPACT compiler, and their work in building up the IMPACT infrastructure made my work possible. Ben-Chung Cheng was always willing to implement the front-end support that I needed, and to help me review my dissertation. David August and Teresa Johnson provided support during the time I was taking classes and preparing for the qualifying examination. Sabrina Hwu and Dan Lavery provided support for me during my time here. Dan Connors, Kevin Crozier, Patrick Eaton, Matt Merten, John Sias, and Le-Chun Wu developed software tools that I needed. I

would also like to thank my officemates for putting up with me — Wayne Dugal, David August, Ben-Chung Cheng, and John Gyllenhaal.

I would like to thank the people at Hewlett-Packard Laboratories. My summer of employment there was invaluable in the development of my technical expertise in the area of compilers and computer architecture. I would like to thank Santosh Abraham, Sadun Anik, Richard Johnson, Vinod Kathail, Scott Mahlke, Bob Rau, and Mike Schlansker.

I would like to thank my other friends who left jobs in industry and decided to go back to graduate school like me. Todd Wey showed me that it was possible to leave the safety of a real job and succeed in your graduate studies. Phil May and Lilly Bondie provided support during the time I was transitioning from life at Motorola to the life of a graduate student.

I would like to thank my mother, Ruth, for providing emotional and financial support to me as I went through this part of my life. I would like to thank my sister and brother-in-law, Debbie and John Rudisill, for their encouragement. I would also like to thank my parents-in-law, Charles and Marcella Weaver, for helping with many home-improvement projects.

Finally, I want to thank my wife, Joanna, for being willing to give up a nice, stable life in warm Arizona, and agreeing to live as a poor student in cold Illinois. She has provided the emotional support that I needed to survive graduate school. Without her assistance, this thesis would never have been written.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

An *instruction-level parallelism* (ILP) compiler uses aggressive optimizations to reduce a program's running time. These optimizations have been shown to be effective when profile information is available. Profile information is useful to guide optimizations [1], [2], [3]. For example, profile information controls code growth by allowing selective application of the optimizations [4] and ensures that the optimizations result in a speedup, not a slow down [5].

To apply aggressive optimizations intelligently, the ILP compiler needs to know relative loop trip counts and the most likely direction of branches for all executions of the program. This information allows the compiler to focus its resources on the important code sections and make the correct trade-offs during optimization. Program profiling gives the compiler this information by using the results of a subset of possible inputs, in order to guide the compilation for the entire program.

Program profiling is accomplished using a *compile-execute-recompile* methodology. The program is compiled with special probes inserted in the executable to measure the program's behavior. The program is then executed with a user-selected set of test inputs, and the run-time behavior is recorded. Finally, the program is recompiled based on the recorded data.

Unfortunately, there are several drawbacks to profiling. First, profiling can be time consuming. This makes it unacceptable for many users. Second, profiling may not be feasible in some environments, such as real-time or embedded applications. Third, profiling accuracy relies on the behavior of the program remaining relatively constant for all possible inputs. If the

program's behavior varies, poor performance after recompilation may occur for some inputs. Finally, profiling must ensure that all important sections of code are executed. Otherwise, the compiler may not optimize the unexercised code sections well, if at all.

For an ILP compiler to be successful, it must be able to do an acceptable job when no profile information is available. This dissertation deals with this issue by investigating non-loop branch prediction and loop-trip-count prediction. Statically estimating this information gives the ILP compiler guidance in the application of optimizations.

In some specific problem domains, static prediction can successfully guide the compiler. For example, many numerically-intensive programs contain loops that always iterate many times or iterate a predetermined number of times, and contain simple control flow. Unfortunately, in general, static prediction is limited because it lacks knowledge about the input data. Many non-loop branches and most loops are directly affected by the input data. This means that the potential performance that can be obtained without profiling the program may be significantly less than the performance that can be obtained when the compiler has profile information available. For this reason, program profiling is desirable.

Even with profile information available, the ILP compiler must ensure that it applies its optimizations intelligently so that performance is not degraded unnecessarily when the program's behavior for a real input differs from the behavior seen during profiling. This can be accomplished by using static analysis to make performance tradeoffs in the compiler's heuristics and by using profile information only when it is necessary. This dissertation proposes the *speculative hedge* acyclic scheduling heuristic, which attempts to minimize the effect of variable program behavior while scheduling superblocks [2] and hyperblocks [3].

The ILP compiler must also be able to effectively compile sections of code that are untouched during profiling. This can be an issue for large programs that contain a large amount of functionality. If users are not careful with their choice of profile inputs, they can leave important code sections untouched during profiling. In addition, certain functionality may be unexercised because the profile time may be unacceptably large. One alternative for the compiler is to use the same techniques for untouched code sections as those used when no profiling is performed because these cases are almost identical. This dissertation evaluates another alternative: evaluate the benchmark as a whole, and predict behavior for unexercised code based on the behavior of code exercised during profiling. This allows the static prediction to benefit from knowledge about the input data gained during profiling.

This dissertation looks at the problems facing ILP compilers when trying to apply aggressive optimizations to real programs. The problems include doing an effective job when program profiling is not performed, not degrading performance needlessly when the program behavior differs from the behavior seen during profiling, and doing an effective job on portions of the program untouched during program profiling. A successful ILP compiler can be realized if these problems are resolved. This dissertation provides the groundwork needed by ILP compilers to effectively deal with these difficulties.

1.1 Contributions

The three major contributions of this dissertation are described below:

- The static prediction of branch direction is investigated and analyzed. Improved heuristics are developed that provide better prediction accuracy than previous methods. These heuristics provide the ILP compiler with information that is necessary when applying

aggressive optimizations. In addition, this dissertation provides insights into the mechanisms that allow these predictions to work effectively.

- A *loop grouping* technique is examined and developed. Counted loops and linked list traversal loops are grouped, so that loops that iterate with the same loop control in different locations of the source code can be identified. These groups are used to statically predict the loop trip counts for loops that are unexercised during program profiling. This allows the information obtained during profiling to apply to more of the program, and enables better decisions to be made by the compiler. Loop grouping also has applications beyond its use in this dissertation.

- The *speculative hedge* acyclic scheduling heuristic is examined and developed. This heuristic provides better superblock and hyperblock schedules by limiting its dependence on profile information. It enables all paths in the scheduling region to be given priority during scheduling. Similar concepts should be applied to other compiler optimizations in order to allow them to rely less on profile information and to ensure that code performance is not degraded unnecessarily when the program behaves differently than expected.

1.2 Overview

This dissertation is composed of eight chapters. Chapter 2 presents an overview of the IMPACT compiler. All the compiler techniques described in this thesis are implemented within the framework of the IMPACT compiler.

An overview of the benchmarks and inputs used in this dissertation are presented in Chapter 3. Because this thesis deals with the issues associated with profiling, the benchmarks and

inputs used for evaluation take on a significant importance. Detailed comparisons are made between the different benchmarks and inputs.

Chapter 4 deals with the problem of compilation when no profiling is performed. This is an important issue to deal with because users may not always be willing or able to perform profiling. A major portion of this chapter deals with static branch prediction. It details the heuristics used for prediction, provides insights into why the heuristics work, and includes results of the prediction success. In addition, the issues involved with static loop-trip-count prediction and static frequency generation are addressed.

A new compiler analysis technique, loop grouping, that allows loops with similar loop control to be grouped together is presented in Chapter 5. The use of the technique on counted and linked list loops is described, and a detailed analysis of the success of the technique is presented. Chapter 6 uses the loop grouping technique to predict loop trip counts for loops that are contained in code that is unexercised during profiling.

Chapter 7 deals with the problems of branch behavior that varies between profiling and real executions. The speculative hedge acyclic scheduling heuristic is presented, which alleviates the compiler's dependence on profile information when scheduling, and ensures that the performance does not degrade unnecessarily when the program behaves differently for different inputs. Similar concepts should be applied to other compiler heuristics to ensure that performance is not degraded unnecessarily when the program behavior varies. Finally, in Chapter 8, conclusions and directions for future work are discussed.

CHAPTER 2

OVERVIEW OF THE IMPACT COMPILER

2.1 The IMPACT Compiler

All of the compiler techniques analyzed in this dissertation are implemented within the framework of the IMPACT compiler. The IMPACT compiler is a retargetable, optimizing C compiler that has been developed at the University of Illinois. A block diagram of the IMPACT compiler is presented in Figure 2.1. The compiler is divided into two distinct parts based on the intermediate representation (IR) used. [1] The highest-level IR, Pcode, is a parallel C code representation that keeps loop constructs intact. The lowest-level IR, Lcode, is a generalized register transfer language similar in structure to most load/store processor architecture instruction sets.

In Pcode, memory dependence analysis [6], [7], [8], loop-level transformations [9], memory system optimizations [10], [11], function inlining [12], and statement-level profiling are performed. In addition, flattening is performed in Pcode so the representation only contains simple `if-then-else` and `goto` control-flow constructs [12]. Much of the source-level information needed for the static analysis used in this dissertation is obtained in Pcode and passed down to Lcode. A description of the information that gets passed down is presented in Section 2.2.

The Lcode IR is divided into two subcomponents, the machine-independent IR, Lcode, and the machine-specific IR, Mcode. These two subcomponents share the same data structures. The difference is that Mcode is broken up so that there is a one-to-one mapping between Mcode

[1] IMPACT does have another IR, Hcode. However, this representation is not important. It is only used as an intermediate step between IMPACT's Pcode and Lcode IRs.

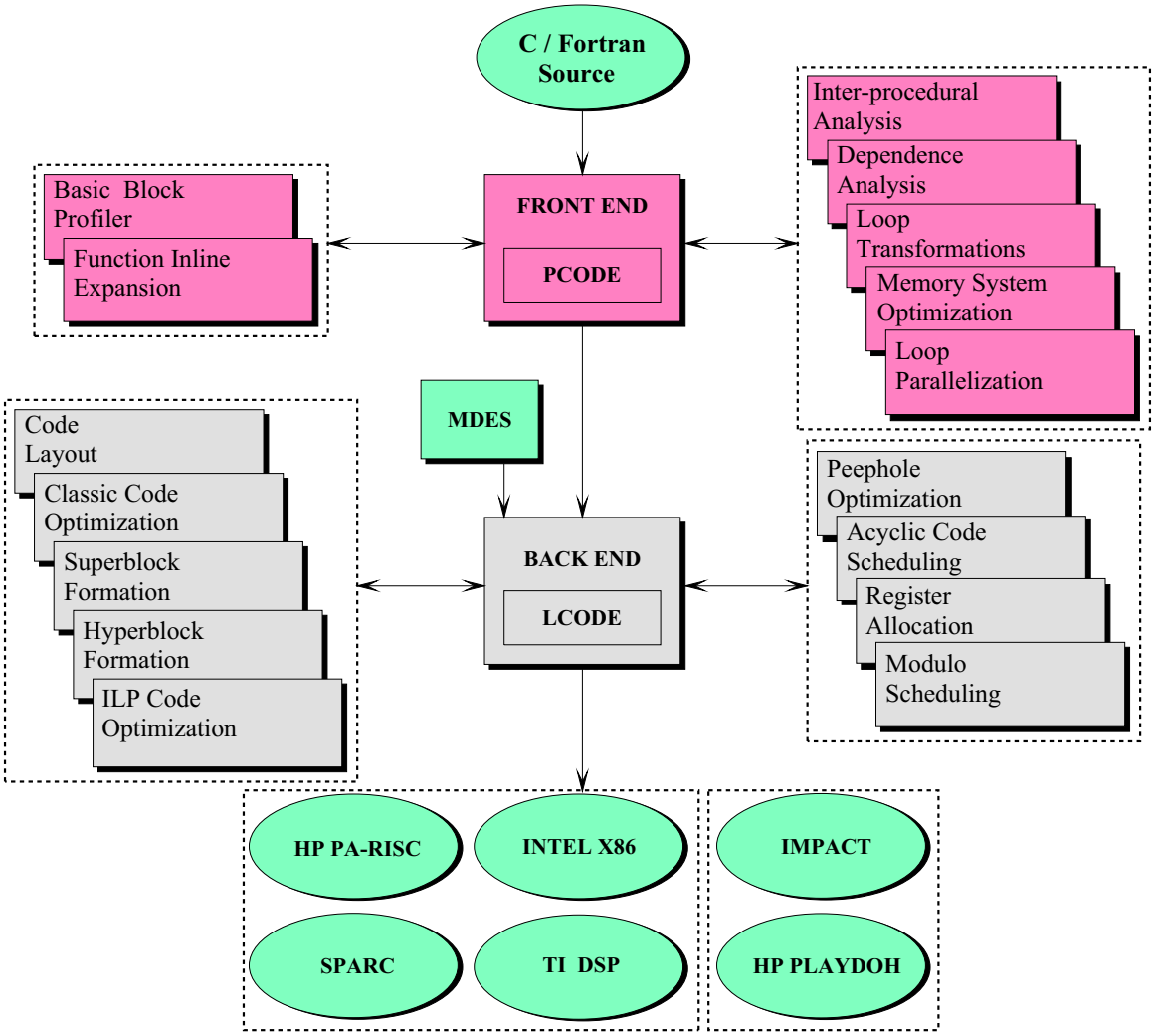

Figure 2.1 Block diagram of the IMPACT compiler.

instructions and the instructions contained in the processor's assembly language. Therefore, the conversion of Mcode to Lcode consists of converting Lcode operations into one or more operations that map to the target architecture. Lcode operations are broken up for a variety of reasons, including limited addressing modes, limited opcode availability, ability to specify a literal operand, and field width of literal operands.

At the Lcode level, all machine-independent classical optimizations are applied [13]. These optimizations include constant propagation, forward copy propagation, backward copy propa-

gation, common subexpression elimination, redundant load elimination, redundant store elimination, strength reduction, constant folding, constant combining, operation folding, operation cancellation, code reordering, dead code removal, jump optimization, unreachable code elimination, loop invariant code removal, loop global variable migration, loop induction variable strength reduction, loop invariant variable elimination, and loop induction variable reassociation. It is after the application of classical optimizations that the static analyses described in Chapters 4, 5, and 6 are performed.

After classical optimization, ILP transformations are applied. These involve the application of superblock code transformation and optimization [2]. When predicated execution support is available in the target architecture, hyperblock techniques [14], in addition to superblock techniques, are used as the underlying compilation structure. In addition, a suite of hyperblock-specific optimizations to further exploit predicated execution support is available.

After the ILP optimizations and transformations are applied, code generation is performed. The two most significant components of code generation are instruction scheduling and register allocation. Scheduling is performed by either global acyclic scheduling [5], [15] or software pipelining using modulo scheduling [16], [17]. Section 7.2 describes the speculative hedge acyclic scheduling heuristic that makes acyclic scheduling less dependent on profile information. In addition to control speculation, both scheduling techniques are capable of exploiting architectural support for data speculation to achieve more aggressive schedules [7], [18], [19].

Graph coloring-based register allocation is utilized for all target architectures [20]. The register allocator utilizes execution profile information if it is available to make more intelligent decisions. For each target architecture, a set of specially designed peephole optimizations is performed. These peephole optimizations allow the removal of inefficiencies incurred during

Lcode-to-Mcode conversion and take advantage of specialized opcodes available in the target architecture.

A detailed machine description database, *Mdes*, for the target architecture is also available to all the Lcode compilation modules [21]. The Mdes contains a large set of information to assist with optimization, scheduling, register allocation, and code generation. Information such as the number and type of available functional units, instruction input/output constraints, addressing modes, and pipeline constraints is provided by the Mdes. The Mdes is queried by the optimization phase to make intelligent decisions regarding the applicability of transformations. The scheduler and register allocator rely heavily on the Mdes to generate efficient, as well as correct, code.

Currently, six architectures are actively supported by the IMPACT compiler. These include the HP PA-RISC, SPARC [22], Intel X86, and the TI digital signal processors, TMS 320C30/C40. The other two supported architectures, IMPACT and HPL PlayDoh [23], are experimental ILP architectures. These architectures provide an experimental framework for compiler and architecture research. The IMPACT architecture is a parameterized superscalar processor with an extended version of the HP PA-RISC instruction set.

2.2 IMPACT Support for Static Analysis

All the compiler techniques described in this dissertation are implemented in the Lcode portion of the IMPACT compiler framework. The optimization and library support found in IMPACT's Lcode modules makes this the best implementation choice, but much of the static analysis presented in this dissertation could have been implemented in Pcode also.

load-name = base-name | **load-name next-level-access**
next-level-access = ϵ | .**abs-offset**_field-name
abs-offset = start_end

Figure 2.2 Grammar for the `ACC_NAME_BY_TYPE` attribute.

Source-level information is necessary for the static analysis in this dissertation to be effective. Since this analysis is being performed in Lcode, source-level information must be passed down from the IMPACT Pcode front-end. This information is used to understand the derivation of branches and to know whether structures and fields are being accessed by loads.

Several annotations are made to branches in Lcode in order to determine where in the source code they are derived. If branches are derived from `if-then-else` statements, the `?` operator, or `switch` statements, they are marked accordingly. In addition, branches that are derived from the same conditional expression of `if-then-else` statements or loops have their relationship marked with a syntax tree [24]. The syntax tree allows the original complex expression to be recreated and the original relationship of all the branches from the same complex expression to be known. Also, branches that compare two pointer values are marked.

Important information is also passed down for loads and is used heavily in the loop grouping technology described in Chapter 5. Information is provided that allows the aggregate data structure and field names being accessed by loads to be known in Lcode. This is done through the use of the `ACC_NAME_BY_TYPE` Lcode attribute. The usefulness of the `ACCESS_NAME_BY_TYPE` attribute comes from understanding that two loads are accessing the exact same structures, unions, and fields. It is assumed that data contained in the same structures, unions, and fields are going to behave similarly. The grammar for this attribute is defined in Figure 2.2.

In this grammar, there are five terminals: base-name, field-name, start, end, and ϵ. The terminal *base-name* is the type of the variable declared in the program from which the load is initiated; *field-name* is the name of the structure or union field that is being accessed, and it allows accesses to different names in a union that are overlapped to be differentiated; *start* is the absolute start address for a field; and *end* is the absolute end address for a field. The nonterminal *load-name* is set to base-name when there is a simple assignment (i.e., there are no structure or union field accesses). The nonterminal load-name is set to "load-name next-level-access" when a structure or union field is accessed. The *next-level-access* is used to define accesses to the next level of a structure or union field, which could be a nested structure or union. Finally, *abs-offset* holds the absolute address range for the location of the current field being accessed relative to the start of the current structure or union. There is redundancy between abs-offset and field-name when the current field is located in a structure.

Examples of the names present in the `ACCESS_NAME_BY_TYPE` attribute are shown in Figure 2.3. Figure 2.3(a) shows the declarations for the variables used in the examples. Figure 2.3(b) shows actual lines of source code and the corresponding attribute that would be located on the last load needed for each source line. Number 1 in the figure is a simple assignment of an integer. Number 2 is a direct access of `field1` in the `s3` structure. Number 3 is an indirect access of `field1`, and requires two loads. The first load is needed to obtain the address of the structure containing `field1`, and it would have an attribute of `example_struct`. Finally, number 4 is accessing the `next` field through two pointers and requires three loads. The first load would have an attribute of `example_struct`, and the second load would have an attribute of `example_struct.4_7_next`.

```
int g1, g2;
struct example_struct {
    int field1;
    struct example_struct next;
} *s1, *s2, s3;
```

(a)

	Source code	Attribute for load
1.	g1 = g2;	int
2.	g1 = s3.field1;	example_struct.0_3_field1
3.	g1 = s1–>field1;	example_struct.0_3_field1
4.	s1 = s2–>next–>next;	example_struct.4_7_next.4_7_next

(b)

Figure 2.3 Examples of the `ACC_NAME_BY_TYPE` attribute: (a) variable declarations, (b) actual examples.

CHAPTER 3

BENCHMARKS

This dissertation provides a base-level technology so that ILP compilers can effectively deal with the difficulties associated with profiling. The goal is for an ILP compiler to generate efficient code even if profiling is not performed or is done poorly. This is a serious issue for an ILP compiler to deal with because many optimizations have been shown to be effective only when profile information is available.

Before going into the details of this work, the benchmarks used in this dissertation are analyzed. This is an important topic because the benchmarks and inputs used in this dissertation are going to illustrate the effectiveness of the techniques needed to deal with the profiling problems.

This chapter is composed of two sections. Section 3.1 introduces the benchmarks and inputs used in this dissertation. Section 3.2 analyzes the chosen benchmarks and inputs, and provides insight into their behavior.

3.1 Benchmarks and Inputs

The benchmarks used during the development of the techniques found in this dissertation come from a set of UNIX utilities; the six SPEC CINT92 benchmarks; the two C programs, *alvinn* and *ear*, from SPEC CFP92; the eight benchmarks from SPEC CINT95; and a set of

Table 3.1 Description of the benchmarks used in this dissertation.

Benchmark	Source	Description
espresso	SPEC CINT92	Boolean function minimizer
sc	SPEC CINT92	Spreadsheet
go	SPEC CINT95	Game of go
m88ksim	SPEC CINT95	Simulator for the 88100 microprocessor
gcc	SPEC CINT95	GNU C compiler version 2.5.3
compress	SPEC CINT95	Data compressor and decompressor
li	SPEC CINT95	Lisp interpreter
ijpeg	SPEC CINT95	JPEG encoder and decoder
perl	SPEC CINT95	Perl interpreter
vortex	SPEC CINT95	Single-user object-oriented database
Pcode	IMPACT-ver 3.0	IMPACT front-end module
Lhppa	IMPACT-ver 3.0	IMPACT HP PA-RISC code generator module

programs from the IMPACT compiler.[1] References to many of these benchmarks are located throughout this dissertation. However, the results in this dissertation are shown on a subset of these benchmarks in order to effectively present the results. These benchmarks are shown in Table 3.1 and include all the programs from SPEC CINT95, as well as two programs from both SPEC CINT92 and the IMPACT compiler.

For each benchmark used in this dissertation, a set of three inputs is used. Three inputs are used because they allow differing behavior found in the benchmarks to be exposed, and using only three inputs ensures that the time needed to analyze all the benchmarks is not too large. The three inputs were carefully chosen from a larger set of inputs, and analysis of the relative behavior of the different inputs was used to make the decisions on which inputs to include. The results present in Section 3.2 were used during this decision process.

[1] The IMPACT code comes from the version 3.0 beta release dated June 10, 1997, which was made available to several universities outside of the University of Illinois.

The set of inputs used for each of the benchmarks is shown in Table 3.2. For the remainder of this dissertation, specific inputs are referenced by the *input number* found in this table. More details on the inputs are presented in Section 3.2.

Many of the inputs used for the SPEC benchmarks are obtained from the inputs provided with these benchmarks. There are three exceptions. For *gcc*, inputs 1 and 2 use the command line options that are to be applied when running the SPEC inputs (these options force a large number of optimizations to occur), while input 3 forces minimal optimization and outputs debugging information. For *li*, the SPEC CINT92 reference input, 9 queens, is used, along with the *tomcatv* rewrite and prime number sieve used by Fisher and Freudenberger [25]. Finally, for *ijpeg*, inputs 1 and 2 perform compression as applied when generating SPEC results, while input 3 does decompression.

The inputs for the IMPACT programs allow a variety of functionality to be exercised. Both *Pcode* and *Lhppa* contain a wide range of functionality. *Pcode*'s inputs 1 and 2 perform internal representation transformations, while input 3 creates summary information used by an interprocedural analysis routine. The *Lhppa* code generator can generate code just like an HP PA-RISC compiler, but it can also generate emulation code in order to perform testing of advanced processor architectures. *Lhppa*'s input 2 generates code for an HP PA-7100 similar to the HP compiler, and inputs 1 and 3 generate emulation code for profiling and simulation of code for an advanced architecture.

3.2 Benchmark Behavior

The behavior of the benchmarks and inputs used in this dissertation is presented in this section. Comparisons are made between the branch and loop behavior for the three different

Table 3.2 Description of the inputs used for each benchmark.

Benchmark	Input Number	Description
espresso	1	bca from reference dataset
	2	tial from reference dataset
	3	Z5xp1 from short dataset
sc	1	loada2 from reference dataset
	2	loada1 from reference dataset
	3	loada3 from reference dataset
go	1	5stone21 from reference dataset
	2	null from test dataset
	3	2stone9 from train dataset
m88ksim	1	reference dataset
	2	test dataset
	3	train dataset
gcc	1	insn-emit.i from reference dataset using all the SPEC command line options
	2	stmt-protoize.i from reference dataset using all the SPEC command line options
	3	insn-emit.i from reference dataset using "-Wall -g" as the command line options
compress	1	reference dataset
	2	test dataset
	3	train dataset
li	1	9 queens reference dataset from SPEC CINT92
	2	SPEC CFP92 program *tomcatv* rewritten in XLISP
	3	prime number sieve
ijpeg	1	compression of specnum from reference dataset
	2	compression of vigo from train dataset
	3	decompression of specnum from reference dataset
perl	1	primes from reference dataset
	2	primes from test dataset
	3	jumble from train dataset
vortex	1	reference dataset
	2	test dataset
	3	train dataset
Pcode	1	Pcode-to-Hcode transformation with flattening of SPEC CINT92 *compress*
	2	annotation of Pcode profiling results into an intermediate version of c_parse.tab.c from SPEC CINT92 *cc1*
	3	generation of summary information for interprocedural analysis of an intermediate version of eval.c from SPEC CINT95 *perl*
Lhppa	1	generation of an HP PA-RISC assembly file for profiling from an unscheduled, ILP-optimized Lcode version of SPEC CINT92 *compress*
	2	generation of an HP PA-RISC assembly file for execution on an HP PA-RISC PA7100 from a classically optimized Lcode version of c_parse.tab.c from SPEC CINT92 *cc1*
	3	generation of an HP PA-RISC assembly file for simulation from a derived version of eval.c from SPEC CINT95 *perl*, which has been compiled down to the assembly level for a hypothetical machine model

inputs of each benchmark. Along with this information, a description of differences between the inputs is given. In addition, analysis of loop-trip-count behavior and the effectiveness of the inputs in guiding the compilation of the benchmarks is evaluated.

The first comparison made between the benchmark inputs is shown in Table 3.3. In this table, comparisons are made between pairs of inputs. There are three possible pairings — 1 versus 2, 1 versus 3, and 2 versus 3 — and the comparison for each row is shown in the *input pairs* column. For this table, the number before the slash in the column is referred to as *1st* input, and the number to the right of the slash is referred to as the *2nd* input.

This table compares how pairs of inputs exercise the branches differently, and comparisons are done for *dynamic branches* (each branch gets counted once every time it is executed during a program execution) and *static branches* (each branch present in the source code gets counted once). In the table, *d* in a column title means that dynamic branch results for a particular input are presented, and *s* in a column title means that static branch results are presented. This evaluation is done on IMPACT Lcode versions of the programs that have been classically optimized. In addition, all `switch` statements are converted into discrete, adjacent branches.

The first major comparison made in Table 3.3 deals with *branch coverage*. The percentage of branches exercised by the first input in the row and unexercised by the second input is presented in the *only 1st %* column. The opposite case, in which the second input exercises branches and the first input does not exercise the same branches, is presented in the *only 2nd %* column. The percentage of branches in the source code that are unexercised by both inputs involved in the row is presented in the *untouched %* column.

The percentage of branches unexercised in the source code for the different input pairings is high. The average unexercised percentage over all the benchmark and input pairings is 50%.

Table 3.3 Benchmark and input branch comparison statistics.

		Branch Coverage			Branch Direction			
	Input Pairs	Only 1st %	Only 2nd %	Untouch %	Change 1st %	Change 2nd %	Bias	Bias
Benchmark	*1st/2nd*	*d/s*	*d/s*	*s*	*d/s*	*d/s*	1st %	2nd %
espresso	1/2	7/1	1/6	51	23/7	20/7	71	70
	1/3	13/3	0/1	56	37/15	16/15	79	83
	2/3	5/7	0/0	52	8/9	7/9	71	79
sc	1/2	6/7	15/1	47	11/6	14/6	60	64
	1/3	6/12	2/1	47	10/6	7/6	67	74
	2/3	1/6	0/1	53	23/7	11/7	75	63
go	1/2	0/2	1/3	3	2/9	2/9	58	56
	1/3	0/11	0/1	5	6/11	5/11	57	55
	2/3	1/13	0/1	4	6/12	5/12	57	56
m88ksim	1/2	5/6	0/0	59	1/4	1/4	67	63
	1/3	1/10	0/0	59	3/5	2/5	71	88
	2/3	1/10	10/6	59	3/9	2/9	91	93
gcc	1/2	0/0	4/25	44	4/6	5/6	65	65
	1/3	38/16	6/2	67	2/4	1/4	68	73
	2/3	55/41	6/1	42	3/6	2/6	72	71
compress	1/2	29/16	0/0	11	16/8	6/8	77	85
	1/3	0/8	0/0	11	21/7	16/7	78	67
	2/3	0/0	2/8	19	3/4	9/4	68	68
li	1/2	2/10	0/0	59	15/8	16/8	72	69
	1/3	3/10	1/3	56	12/10	16/10	79	70
	2/3	5/3	3/6	63	15/8	14/8	85	71
ijpeg	1/2	0/0	2/1	65	0/1	0/1	81	82
	1/3	10/3	16/10	55	29/4	22/4	88	76
	2/3	10/4	16/10	54	25/5	22/5	87	76
perl	1/2	0/0	0/0	75	4/1	2/1	78	60
	1/3	6/4	19/9	66	7/7	10/7	84	86
	2/3	16/4	19/9	66	8/7	9/7	82	87
vortex	1/2	0/0	0/0	37	0/1	1/1	64	73
	1/3	0/0	0/0	37	0/1	1/1	68	66
	2/3	0/0	0/0	37	0/1	0/1	58	65
Pcode	1/2	4/4	10/3	78	2/5	10/5	63	96
	1/3	6/7	99/18	63	3/7	18/7	66	80
	2/3	11/6	99/17	64	2/6	1/6	61	59
Lhppa	1/2	5/1	55/9	81	4/7	3/7	72	69
	1/3	7/3	36/2	88	2/4	1/4	65	69
	2/3	56/11	39/3	79	7/7	10/7	59	91

In addition, for 9 out of the 12 benchmarks evaluated, at least one input exercises less than 50% of the branches derived from the source code. This occurs because many of the programs contain a large set of functionality, and one input only exercises a small part of it. For *Pcode* and *Lhppa*, two of the programs with the highest untouched percentages, extensive library support is present, and this support is never fully utilized by a single input.

The second major comparison made in Table 3.3 deals with *branch direction*. The branch direction for branches that are exercised by both inputs is compared. If branch directions for the two inputs are opposites (i.e., one input has the branch taken greater than 50% of the time and the other input has the same branch taken less than 50% of the time), the results get presented in the *branch direction* columns. The percentage of branches involved with the first input is shown in the *change 1st %* column, and the percentage of branches involved with the second input is shown in the *change 2nd %* column. Since the same branches are involved with both of these columns, the static branch percentages in both columns are identical.

The last two columns, *bias 1st %* and *bias 2nd %*, show the percentage of time that the dynamic branches go in the direction predicted by profiling. They show the bias of the branches for going in a particular direction. If the bias percentages are close to 50%, the difference in branch predictions between the inputs does not matter much. (That is, if one input predicts that a branch is taken 49% of the time and the other input predicts that a branch is taken 51% of the time, the branch direction based on profiling differs, but the actual difference in percentages is small.) If the bias percentages are close to 100%, the difference in branch predictions is significant because the two inputs are forcing different behavior in the branch.

In the *branch direction* columns, a large portion of the differences occur for two reasons. First, the average loop trip counts cause differing branch behavior for loop back-edge branches

when one input forces the average trip count to be less than two, and the other input forces the average trip count to be greater than two. A trip count of two is important because loop back-edge branches are located at the end of the loop body, and as long as a loop iterates at least twice, the loop back-edge branch is taken the majority of the time. However, if the average loop trip count is less than two, the loop back-edge branch is not taken the majority of the time. In fact, if the average trip count for a loop is one, the loop back-edge branch is never taken. Second, `switch` statements in this analysis are converted into adjacent branches, not into indirect jumps. If one of the `case` statements is taken the majority of the time by one input and not taken as often by the second input, the branch direction associated with that one particular `case` statement will differ between the two inputs.

Now, the effect of the different inputs on the branch behavior is analyzed in more detail. This analysis provides more insights into the behavior of the different inputs used for each benchmark.

espresso – The major difference between input 1 and the other two inputs concerns the use of the binary variables functionality, as found in the `cube_struct` data structure defined in the file, espresso.h. Input 1's data use this functionality, but the data used for inputs 2 and 3 do not exercise this same functionality. Another difference is that input 3 does not contain a "-" in its input data, while the other two inputs contain this symbol.

sc – The major difference between the inputs concerns the computations that are performed on the spreadsheet. Inputs 2 and 3 only perform a sum, while input 1 performs every other computation except a sum — these computations include a product, average, standard deviation, maximum determination, and minimum determination. The differences in the instruction mix for each input does change the importance of individual `case` statements,

and the differing number of rows and columns used by the inputs also causes the program behavior to vary.

go – The input differences concern the playing level, board size, and starting position for the game. However, these differences do not make a major difference in the program behavior.

m88ksim – The major difference between the inputs is that input 2, which is the test input provided with this SPEC benchmark, does not enable the cache simulation.

gcc – There are several differences between the inputs. The major difference is that input 3 compiles using the "-g" option, while the other two inputs use the normal SPEC "-O" options. There is also a difference in the C statements that are utilized in the input files. Inputs 1 and 3 use the input file, insn-emit.i, which generates instruction sequences, and does not use a wide variety of C constructs. Input 2 uses the input file, stmt-protoize.i, which uses enumerated types and `switch` statements that are not present in insn-emit.i.

compress – The difference between the behavior of the different inputs comes from the dataset size used in the input. Input 1 has the largest dataset; input 3 has a much smaller, yet fairly large dataset; and input 2 uses the smallest dataset. The different dataset sizes require differing amounts of support when handling conflicts in the hashing tables used in *compress*. The dataset for input 2 is so small that there are no conflicts, and a loop in *compress* that is traversed many times when there are conflicts is avoided.

li – The differences come from the XLISP programs that are used. The differences in branch directions are mostly attributable to loops that iterate over the `cdr` set. The same loop for one input iterates few times (less than two), while another input iterates over the same loop many times.

ijpeg – The differing behavior of the inputs results from the fact that compression is being done by inputs 1 and 2, and decompression is being done by input 3. This causes different sections of code to be exercised. There is almost no difference in the behavior of inputs 1 and 2, where the only difference between them is the image that is being compressed.

perl – The difference in performance comes from the different Perl scripts that are used. Inputs 1 and 2 uses the script primes.pl, which determines by brute force if the number it is given is a prime number. Input 3 uses the script jumble.pl, which rearranges scrambled words to find words contained in its dictionary. These programs force different functionality in *perl* to be executed depending on the script that is being executed. For the same script and different input data, the behavior of the program is similar.

vortex – The different SPEC inputs provided with *vortex* do not significantly change the program's behavior. The major difference between the inputs is the size of the dataset that gets applied during program execution.

Pcode – Two factors cause the behavior to differ. First, the functionality that is performed for the three inputs is varied. Input 1 performs an internal representation change from IMPACT's Pcode representation to IMPACT's Hcode representation, and it also flattens the code [12]. The only functionality of input 2 is to annotate the results of profiling back into the IMPACT's Pcode representation. Input 3 performs a much more computationally expensive function by generating the summary information necessary for performing interprocedural analysis. Input 3 ends up spending most of its computation time in code that is unexercised by inputs 1 and 2.

Table 3.4 Definition of loop categories used in this dissertation.

Start		Category		End
0	$<$	Low	$\leq$	5
5	$<$	Medium	$>$	20
20	$\geq$	High	$>$	∞

Second, the files that are processed have different characteristics. IMPACT's Pcode is a hierarchical representation that uses a list representation for file I/O. The different file characteristics cause input 1 to average one element per list, input 2 to average 116 elements per list, and input 3 to average 15 elements per list. Therefore, loops that iterate over these elements act differently for the different inputs.

Lhppa – The *Lhppa* benchmark is similar to the *Pcode* benchmark because it has a wide range of functionality. This variation causes the difference in branch behavior seen in Table 3.3. Input 1 generates code for emulation, using a simple register allocation scheme that spills all the registers, and it is the only input that does not contain any `switch` statements. Input 2 performs code generation for the HP PA-7100 processor and is the only input that does not act on superblocks (it only works on basic blocks). Input 3 generates code for emulation and uses the normal register allocation scheme when generating the code.

The loop behavior for the different benchmarks and inputs is now examined. For this dissertation, loop trip counts are classified into low, medium, and high trip count categories. The definitions for these trip count categories are presented in Table 3.4. These categories are important because an ILP compiler will perform different optimizations based on the observed trip count. For a low trip count loop, the latency needs to be minimized; for a high trip count loop, the throughput needs to be maximized.

The first part of the loop analysis determines how the behavior of loops changes for the different inputs. Figure 3.1 shows how the loops behave for the different inputs. The x-axis for the figure contains the benchmarks and inputs. The left bar for each benchmark represents the behavior for input 1, the middle bar represents the behavior for input 2, and the right bar represents the behavior for input 3. The y-axis is normalized so that the fraction of the total *dynamic iterations* taken during a program execution is presented. A dynamic iteration is defined to be one iteration of a loop taken during a program execution. Each time a loop iterates during program execution, it gets one added to its total dynamic iteration count. All the dynamic iteration counts taken for every loop in the program get summed together, and this is the basis for normalization.

In Figure 3.1, the dynamic iterations are put into three categories. These categories represent the cases where the same loop is exercised by multiple inputs. If the average loop-trip-count category (i.e., small, medium, or large) for a loop that is exercised is the same for all the inputs, the loop is contained in a *single* loop-trip-count category. However, if one input gives a loop an average trip count of low or medium and another input gives the same loop a different, yet adjacent average trip count (medium or high), then the loop is contained in a set of *two adjacent* loop-trip-count categories. Finally, if an input gives a loop an average trip count of low and another input gives the same loop an average trip count of high, then the loop is contained in *all three* loop-trip-count categories. The dynamic importance of the loops found in these three categories is shown for each of the benchmarks' inputs.

As seen in the Figure 3.1, the SPEC benchmarks exhibit relatively consistent loop-trip-count behavior among the different inputs. The benchmark *sc* exhibits the most difference, but has less than half of its dynamic iterations in loops that traverse two adjacent loop-trip-count

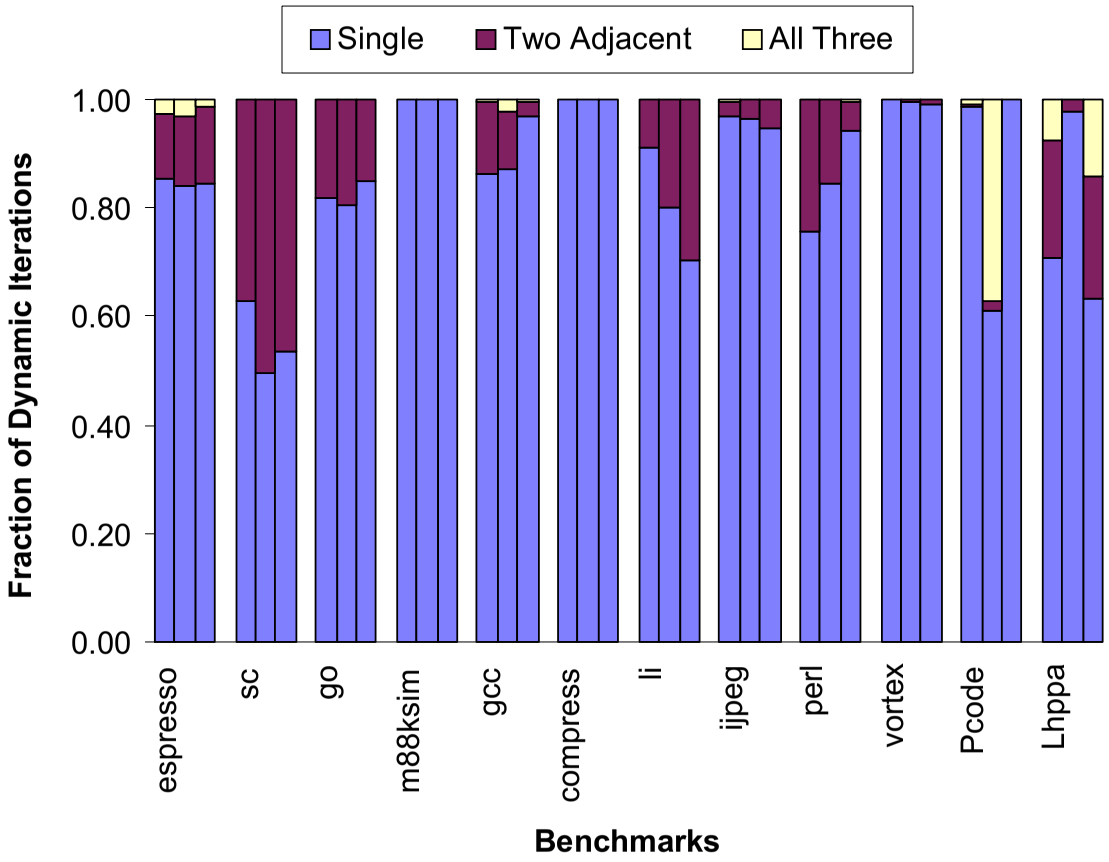

Figure 3.1 Categorization of loops based on differences in average trip count behavior.

categories, and none of the loops traverse all three categories. Most of the loops in *sc* involved with the two adjacent categories occur because of the varying number of rows and columns in the spreadsheet computations that are utilized by the different inputs.

The two IMPACT benchmarks do exhibit varying loop-trip-count behavior across the different inputs. The *Pcode* benchmark's difference, which is described earlier in this section, occurs because the data contained in the different inputs is fundamentally different. Because of this fundamental difference, the average trip counts for the loops in question are 1, 116, and 15 for inputs 1, 2, and 3, respectively. In addition, the amount of computation, which is also described earlier in this section, is different for the three inputs. Input 2 does the least amount

of computation (it is only annotating the results of profiling back into the IMPACT representation), input 1 is doing more computation, and input 3 is so computationally expensive that the "problem" loops do not account for a significant amount of the dynamic loop iterations.

Most of the *Lhppa* loops that exhibit varying trip count behavior across the different inputs occur when traversing the operations contained in the control blocks (control blocks are either basic blocks, superblocks, or hyperblocks, and optimization and scheduling are performed on them). Input 2 works only on basic blocks, while the other two inputs work on superblocks. This means the average trip count for many of these loops in Lhppa is small for input 2, but high for the other inputs.

The second part of the loop analysis determines the distribution of dynamic loop iterations that are spent in each of the loop-trip-count categories. The results of this analysis are presented in Figure 3.2. This figure shares the same x-axis and y-axis definitions used in Figure 3.1. The three categories that are used in this figure match the categories defined in Table 3.4. All the dynamic loop iterations for a particular loop go into the category that matches the average trip count observed for the loop.

As seen in the figure, the dynamic iterations for different benchmarks are dominated by loops with varying average loop trip counts. The benchmarks *go* and *li* are dominated by short trip count loops, and the benchmarks *m88ksim*, *ijpeg*, and *perl* have all their inputs dominated by high trip count loops. Domination by short trip count loops means that most of the loop iterations are spent in short trip count inner loops.

There are even variations between inputs for the same benchmark in terms of the trip counts that dominate execution. Input 2 of the benchmark *compress* is dominated by high trip count loops because its dataset is so small. Since its dataset is small, the input does not exercise any

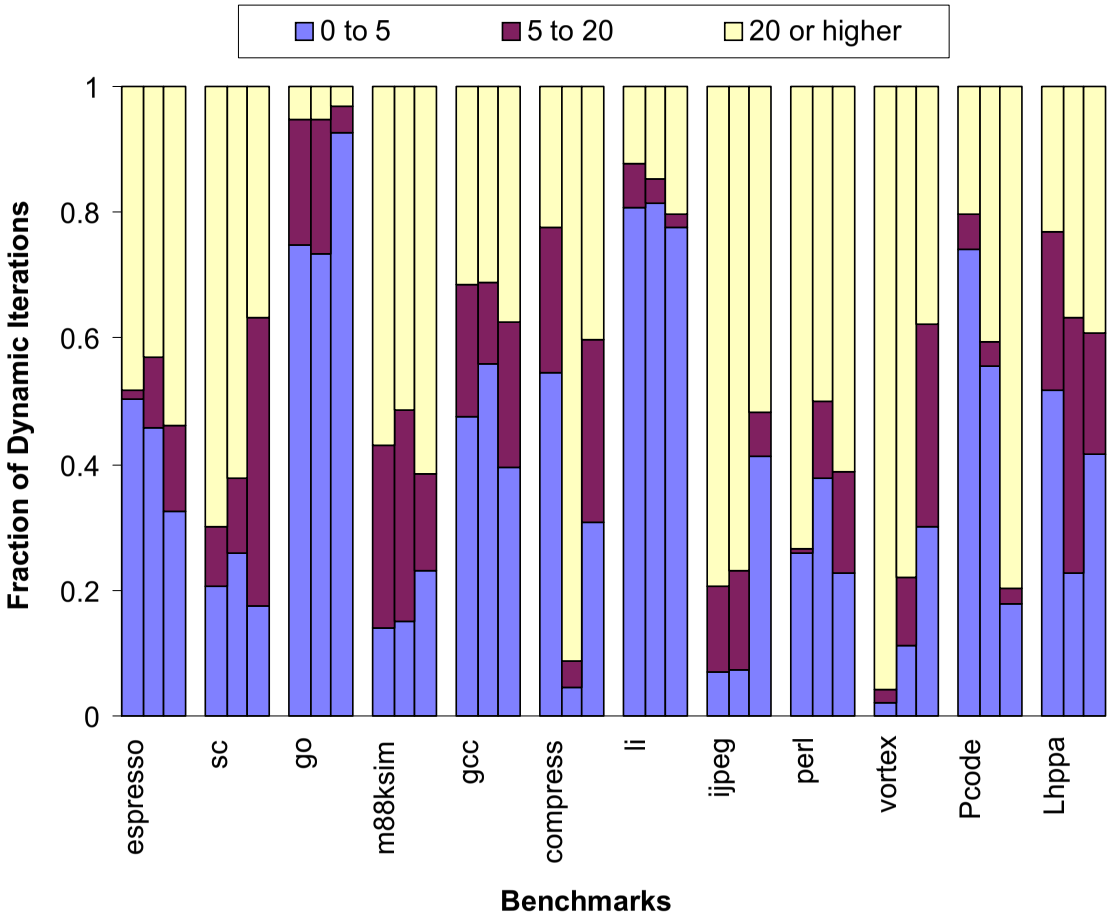

Figure 3.2 Dynamic iterations spent in loops with different average trip counts.

of the short trip count loops needed to handle hashing table conflicts. The benchmark *ijpeg*'s differences come from performing compression and decompression. Compression is performed in inputs 1 and 2, and decompression is performed in input 3. The benchmark *vortex*'s varying behavior is the result of the size of the datasets. The larger the dataset, the smaller the percentage of time that is spent initializing. Initialization is dominated by short trip count loops, while steady-state processing is dominated by high trip count loops. Finally, *Pcode*'s variations are a function of the computations and input data. Input 3 exercises different functionality than the other two inputs, and this functionality is dominated by high trip count loops.

Now the ability for the different inputs to guide the compilation for the other inputs is investigated. This experiment allows one input to guide the compilation. Then, the other two inputs are applied individually, and a measurement is taken of the percentage of time spent in code that was unexercised during profiling. The Lcode that this experiment uses has been aggressively inlined, classically optimized, scheduled, and register allocated, and no ILP optimizations have been applied. The target processor is a four-issue, restricted functional unit processor that includes two integer ALUs, two memory units, one floating-point ALU, and one branch unit. The processor includes 64 registers, and the latencies of the processor match those of an HP PA-7100 processor.

Figure 3.3 shows the results of this experiment. The x-axis includes the six different combinations of profile guiding and input application that can occur when using the three different inputs for each benchmark. The leftmost bar in each benchmark corresponds to the first entry in the legend (*#1 guided- #2 applied*), the next bar to the right corresponds to the second entry in the legend, and so on. The y-axis shows the percentage of time for the applied input that is spent in code that is unexercised by the input used to profile the program. For example, in *gcc*, input 2 spends over 70% of its execution time in code that is unexercised by input 3 when input 3 is used to guide profiling. This corresponds to the *#3 guided- #2 applied* bar in the figure.

In Figure 3.3, *go* and *vortex* do not have any significant amount of time spent in code that is unexercised during profiling. Five benchmarks, *gcc*, *ijpeg*, *perl*, *Pcode*, and *Lhppa*, contain at least one input that spends at least 30% of its execution time in code that is unexercised by another input. The *gcc* unexercised percentages occur because input 3 uses "-g" compilation, while the other two inputs use "-O" compilation. The benchmark *ijpeg*'s unexercised

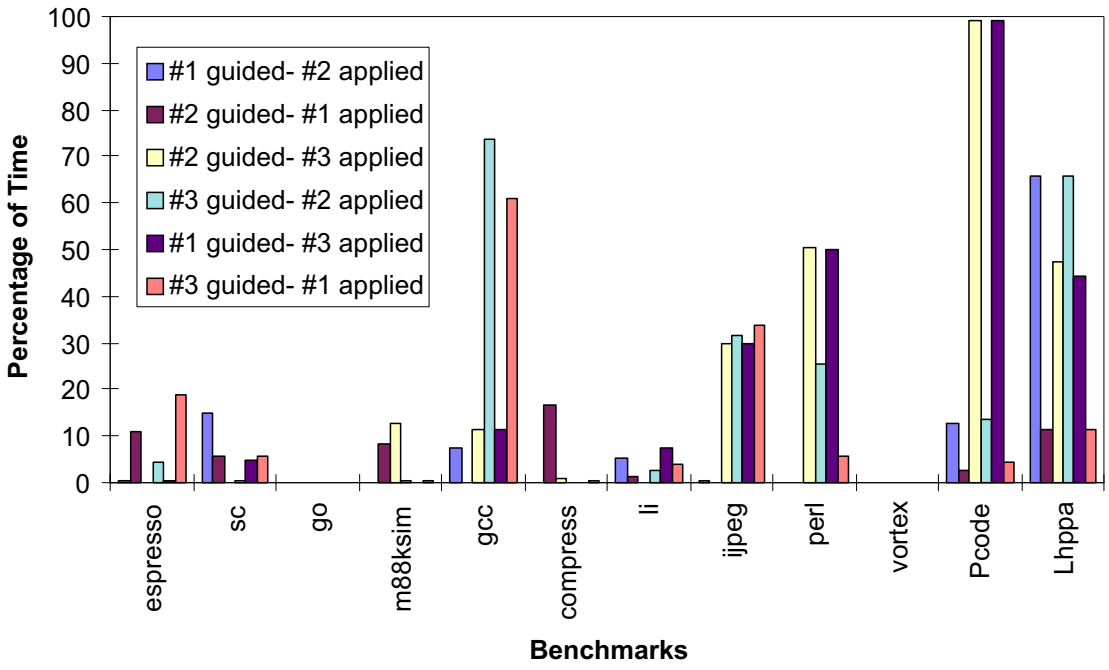

Figure 3.3 Percentage of time spent executing unexercised basic block code.

percentages occur because input 3 performs decompression and the other two inputs perform compression. The use of the primes functionality in inputs 1 and 2, and the jumble functionality in input 3 accounts for the unexercised percentages in *perl*. *Pcode*'s results occur because input 3, which is generating interprocedural summary information for further processing, spends most of its execution time in code that is unexercised by the other inputs. Finally, *Lhppa*'s input 1 does not perform register allocation, and inputs 2 and 3 end up spending a significant amount of processing time performing register allocation.

Many of the benchmark problems discussed in the previous paragraph are contrived and should be avoided during the software development process. For example, *gcc* should not be profiled with "-g", and *ijpeg* should be profiled while performing both compression and decompression. However, it is desirable for an ILP compiler to able to deal with these problem situations because poorly chosen inputs are always a possibility.

Another issue that needs to be considered is the time necessary for profiling. In *Pcode*, the time necessary to execute input 3 is much larger than the time necessary to execute inputs 1 and 2. Because of the significant execution time necessary for execution of input 3, its use for profiling is undesirable. Software developers would benefit from an ILP compiler that can effectively deal with the compilation of unexercised code so that large execution times during profiling can be avoided.

It is also important to realize that the percentage of unexercised execution time present in Figure 3.3 will increase when ILP optimizations are applied. ILP optimizations are effective for code that is exercised during profiling and should make the exercised portion of the program more efficient. These optimizations will not be as effective in making the unexercised part of the program more efficient, which will increase the percentage of time spent in unexercised code.

CHAPTER 4

COMPILATION WITHOUT PROGRAM PROFILING

As mentioned in the introduction, program profiling has been shown to be useful when an ILP compiler aggressively applies its optimizations. Profiling provides the basic block frequency and branch direction information needed by the compiler's optimizations. Without this information, an ILP compiler's ability to generate code that executes efficiently is hampered.

Unfortunately, program profiling is not always performed. Profiling can be time consuming, and this can make it unacceptable for many users. In addition, some environments, like real-time and embedded applications, do not lend themselves to the gathering of profile information. This means that a commercial ILP compiler must be able to deal with the situation when profiling is not performed. The ILP compiler may not be able to generate code as efficiently as it would have if profile information was available, but it should generate code as efficiently as possible.

This chapter looks at static analysis methods that allow the compiler to infer the basic block frequency and branch direction information that is normally obtained via profiling. Section 4.1 provides an introduction into the static analysis techniques evaluated as part of this dissertation. Section 4.2 discusses the static prediction of branch direction, Section 4.3 discusses the issues associated with loop-trip-count prediction, and Section 4.4 touches on the problems of statically generating program frequencies. Finally, Section 4.5 provides a summary of the issues present when trying to statically infer the information normally obtained by profiling.

4.1 Introduction

In order for an ILP compiler to work well when no profiling is performed, the compiler should be able to statically infer basic block frequency and branch direction information. This information can then be used to selectively apply ILP optimizations. The compiler should be aggressive with its optimizations when it is confident of how the program will behave during execution, and much less aggressive when it is not sure how the program will behave. This chapter looks into different methods of static analysis that allow the compiler to understand how a program will behave when it is executed.

Section 4.2 is dedicated to the static prediction of branch direction, with emphasis on non-loop branches. Non-loop branches are important to evaluate because they are difficult to predict statically, and they are important for the effective application of many ILP optimizations. The work in this dissertation discusses heuristics used previously, presents new insights and heuristics for static branch prediction, and evaluates the effectiveness of the heuristics.

The issues with statically predicting loop trip counts are presented in Section 4.3. For general-purpose programs, loop-trip-count prediction is difficult. The work in this dissertation covers the issues that make loop-trip-count prediction difficult, and presents ideas for some heuristics that could be developed to predict the trip counts for certain loops.

The issues with the generation of static frequencies to guide the compiler are discussed in Section 4.4. Static frequencies are a means of presenting the results of static branch prediction and loop-trip-count prediction to the compiler. This method of presenting the results is desirable because frequencies are the common means of presenting the results of profiling to the compiler. This dissertation discusses work that has been done in the past and problems that have been encountered when generating static frequencies.

4.2 Branch Prediction

This section is dedicated to the static prediction of branch direction. There has been a large amount of previous work in this area, and an overview of this work is presented in Section 4.2.1. The heuristics used in this dissertation are described in Section 4.2.2. These heuristics improve upon the work of Ball and Larus [26] and provide new insights into the reason why many of the heuristics work so well. Finally, the success of the static branch prediction heuristics is evaluated in Section 4.2.3.

4.2.1 Previous work

Early static branch prediction techniques were aimed at providing branch predictions for use by the hardware. These techniques employed several different yet simple heuristics that included assuming all branches were taken, assuming backward branches were taken and forward branches were not taken, and assuming certain branch opcodes were taken while other opcodes were not taken [27], [28]. Bandypadhyay et al., in a C compiler for the CRISP microprocessor, assumed branches in loop expressions were resolved so that the program stays in loops, and used a table lookup based on the branch opcode and operand types to determine the direction for non-loop branches [29]. They claimed that they could produce good results, but provided few details.

A different approach to static branch prediction was proposed by Ball and Larus [26]. In addition to predicting that loop branches were resolved so that the program stays in loops, they developed new heuristics to handle non-loop branches. Their major contribution was analyzing the code contained in the two targets of a branch, taken and fall-thru, in order to predict a

Table 4.1 Summary of Ball and Larus heuristics.

Heuristic Name	Heuristic Description
Loop Branch	Predict that an edge back to a loop's head is taken.
Loop Exit	Predict that an edge exiting a loop is not taken.
Pointer	If a branch compares a pointer to NULL or compares two pointers, predict that the pointers are not equal.
Call	Predict that the successor that contains a subroutine call and does not postdominate is not taken.
Opcode	Predict that a comparison of an integer less than zero, an integer less than or equal to zero, or two floating point values being equal fails. Predict that a comparison of an integer greater than zero, or an integer greater than or equal to zero succeeds.
Return	Predict that the successor that contains a return is not taken.
Store	Predict that the successor that contains a store and does not postdominate is not taken.
Loop Header	Predict that the successor that is a loop header or loop preheader and does not postdominate is taken.
Guard	If a branch comparison contains an operand and that operand is used in a successor, predict the successor is taken.

branch direction. For example, they determined that a subroutine call located in only one branch target was usually avoided.

The Ball and Larus heuristics are shown in Table 4.1. The first two heuristics, *loop branch* and *loop exit*, are associated with loops and are predicted so that the loop continues to iterate. The remaining heuristics apply to non-loop branches, where predicting branch direction is much more difficult than for loop branches. In order to predict a branch direction, the heuristics are applied in a certain order. The Ball and Larus application order corresponds to the ordering present in Table 4.1. The first heuristic that applies determines the branch direction (e.g., if the *pointer* heuristic applies, no other non-loop branch prediction heuristic is used).

Wu and Larus suggested a different method for predicting branch direction [30]. Instead of using an application order, they used all the heuristics that applied simultaneously to predict

a branch's direction. They used the hit rates presented in the original Ball and Larus work, and combined them using Dempster-Shafer theory of evidence [31]. The intuition behind this strategy is that the more heuristics that predict the same direction, the more likely the branch direction is going to be correct; and the more the heuristics differ in their predictions, the less likely the heuristics are going to be correct. Calder et al. showed that this prediction scheme actually performs worse than the more *ad hoc* application order used by Ball and Larus [32]. This is likely the result of the dependencies between the different prediction heuristics, which goes against the strong independence assumptions present in Dempster-Shafer theory.

Several other papers have built upon the work of Ball and Larus. Hank et al. used static branch prediction to guide superblock formation [33]. Since static superblock formation was their focus, they were actually more concerned with the avoidance of hazards than predicting branch directions. Wagner et al. used a subset of the Ball and Larus heuristics, but applied them to an abstract-syntax tree representation [34]. Calder et al. trained a neural net to perform static branch prediction and used neural net inputs that were similar to those used in the Ball and Larus heuristics [32]. Finally, Patterson extended static branch prediction by propagating the possible values that operands could have at any time [35]. This allowed accurate branch prediction to occur for branches located in a `for` loop with known loop bounds.

The static branch prediction work in this dissertation also builds upon the work of Ball and Larus. This work provides new intuitive understanding into the reasons why certain heuristics perform so well. In addition, this work is geared toward use in a compiler. Most of the previous work has been performed on program traces, and it has not emphasized the use of static branch prediction in the compiler. Finally, this work improves the state of the art in static branch prediction by improving several heuristics and proposing other new heuristics.

4.2.2 Heuristics

This section describes the heuristics for static branch prediction that are used in this dissertation. These heuristics have been developed through analysis of branch behavior for the benchmarks and inputs described in Chapter 3, and through understanding the original location of branches in the source code. These heuristics improve the state of the art in static branch prediction.

The static branch prediction heuristics benefit from source-level and type information. This information includes knowledge of the location of a branch in the source code (e.g., is a branch derived from a conditional expression in an `if` statement or loop?), and knowledge of the branch operand types (e.g., is one of the branch operands a pointer?). For the most part, this information was not used in previous static branch prediction work. It was not available to most of the previous researchers because they were performing static branch prediction on program traces, and they did not have access to source-level information.

Static branch prediction in this dissertation is conducted in IMPACT's back end, called Lcode. The source-level and type information needed for the static branch prediction is passed down from IMPACT's front end, called Pcode. Performing the analysis in Lcode is done for two reasons. First, the optimization infrastructure and library support are much better in Lcode than in Pcode. The optimizations allow branches that are resolvable at compile time to be removed, and they allow branches that are contained in dead code to be removed. This ensures that static branch prediction is only performed on branches that have unknown behavior at compile time. In addition, the library support in Lcode allows easy application of analysis techniques such as loop detection, dominator control-flow analysis, and reaching definition data-flow analysis. Second, the compiler front end can be bypassed when compilation

is performed on code that has been binary translated. In this case, the compiler needs the ability to perform static branch prediction in the back end of the compiler. Even though the static branch prediction is done in Lcode, there is no reason why the Pcode environment could not be improved to provide the optimization and library support needed to successfully perform the same static branch prediction.

The following subsections describe the heuristics of this dissertation. Each subsection starts with the definition of the heuristic and is followed by a discussion of the heuristic. For each heuristic, a comparison to the similar Ball and Larus heuristic is made.

4.2.2.1 Loop branches

> Predict branches so loops iterate. When a branch is derived from a conditional loop expression, predict that the edge back to a loop's head is taken. For any branch, predict that an edge exiting a loop is not taken.

Loop branches are predicted so the branch direction that allows the loop to continue to iterate is taken. Predicting the direction of loop branches is not difficult. As long as a loop iterates at least twice, these branches should be predicted correctly. A much tougher problem is to statically determine the trip count for a loop. The issues associated with static loop-trip-count prediction are covered in Section 4.3.

Loop branches can be of two types: *loop back-edge* branch and *loop exit* branch. Loop branches are located in loops, and these loops can be determined using natural loop analysis [24] on a control flow graph. A loop back edge is defined to be a control flow arc whose head and tail are contained in the same natural loop, and the head dominates the tail. A loop exit is defined to be a control flow arc whose tail is contained in the natural loop and whose head is not contained in the natural loop. Ball and Larus defined a loop branch to be a branch where one of the outgoing edges is a loop back edge or loop exit. They defined a non-loop branch to

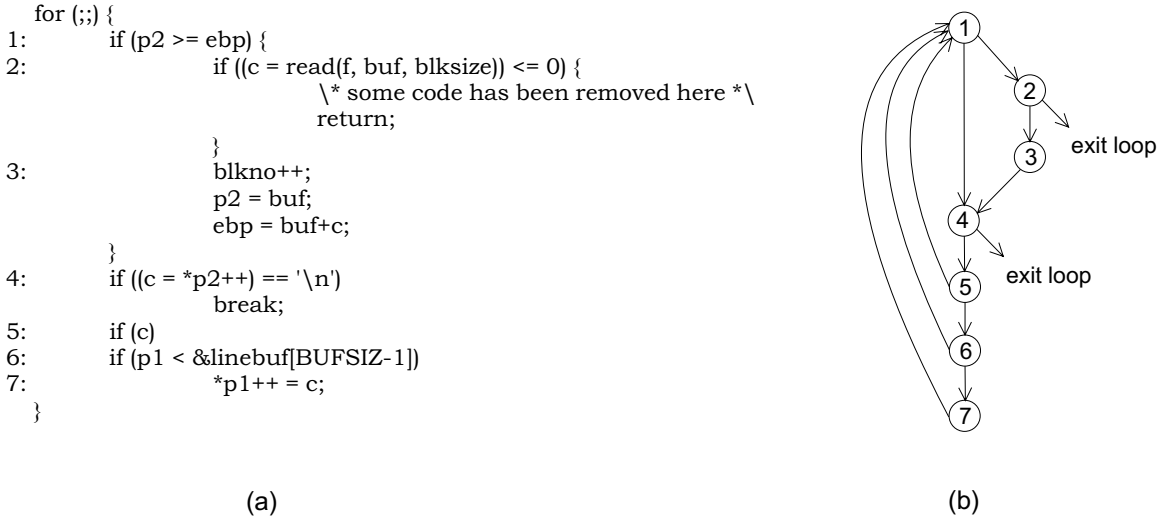

Figure 4.1 Example of a potential problem for the loop back-edge heuristic from *grep*'s function *execute*: (a) source code, (b) control flow graph.

be a branch where neither of the outgoing edges is a loop back edge or loop exit [26]. In this dissertation, similar definitions apply. However, there is one exception made for a situation involving loop back edges. A description of this special case follows.

The special case for loop back edges is needed because a few loop back-edge branches have absolutely no control over the number of times that a loop iterates. This is illustrated by an example from the program *grep* in Figure 4.1. In Figure 4.1(a), the source code for the loop of interest is shown. The numbers to the left of the source code are used to uniquely identify the basic block leaders contained in the loop. The loop's corresponding control-flow graph is shown in Figure 4.1(b). The number in each node of the control-flow graph matches the basic block number located in the source code.

As seen in Figure 4.1(b), there are three back edges contained in this loop: 5→1, 6→1, and 7→1. The two back edges exiting nodes 5 and 6, which are associated with conditional branches, have no control over the continued iteration of this loop. If they are not taken, the unconditional branch located at the end of node 7 ensures that the loop continues to iterate.

In fact, if the source code for basic blocks 5, 6, and 7 is swapped with the `if` statement in basic block 4 and its corresponding `break` statement, there would be no loop back-edge arcs exiting nodes 5, 6, and 7, and the non-exit arc from node 4 would be the only loop back-edge arc contained in this loop.

In this loop, the branches associated with nodes 5 and 6 are only guarding the execution of the statement contained in node 7. Since they are not necessary in controlling the iteration of the loop, they should not be considered a loop branch for the purposes of branch prediction. This solves a major misprediction problem in *grep*. The two branches associated with nodes 5 and 6 account for 27%, 31%, and 47% of the dynamic branches encountered when three different inputs are applied to *grep*. For these three inputs, these branches are mispredicted 100%, 77%, and 53% of the time.

The solution to this problem is to not allow a branch derived from an `if` statement or `?` operator in the source code to be predicted using the loop back-edge heuristic. Only branches located in a conditional loop expression are allowed to have the loop back-edge heuristic applied to them.

This problem does not occur in loops containing conditional expressions. It does not show up in these loops because of the way that most compilers transform loops. This transformation is illustrated in Figure 4.2. Many compilers, including IMPACT, transform `while` and `for` loops by replicating the conditional loop expression in a newly created `if` statement, using the `if` statement to guard the loop's execution, and transforming the original `while` or `for` loop into a `do-while` loop. The advantage of this approach is that an unconditional branch can be removed from the loop.

```
if (j) {
    for (i = init; i < loop_bound; i++) {
        do_something;
    }
}
```

(a)

```
if (j) {
    i = init;
    if (init < loop_bound)
        do {
            do_something;
            i++;
        } while (i < loop_bound);
}
```

(b)

Figure 4.2 Example of a loop code transformation performed by the compiler: (a) loop in the source code, (b) loop after the compiler transformation.

Since the conditional loop expression is located at the end of the loop, all `continue` statements must have a target at the end of the loop. In the *grep* example shown in Figure 4.1, there is no conditional loop expression, so the branches in the `if` statements in question can target the top of the loop directly. If a `while` loop does not get transformed into an `if` statement guarding a `do-while` loop, any branch with a `continue` statement as one of its targets will have a situation similar to the one seen in *grep*.[1]

4.2.2.2 Call

> If the successor contains a subroutine call and does not postdominate, predict that the successor is not taken. If the subroutine call is associated with I/O buffering, exiting, error processing, memory allocation, or printing, a much higher confidence in the prediction is possible.

The Ball and Larus call heuristic predicts that a successor containing any subroutine call that does not postdominate is not taken [26]. The reason they gave for this heuristic working

[1] As long as the `for` loop has a statement in the incrementing portion of the `for` statement, it will not have a similar problem. This happens because the incrementing statement would be located at the end of the loop, and all `continue` statements would have the incrementing statement as their target.

well is that many conditional calls are used to handle exceptional conditions. While this often true, not all subroutines executed conditionally are involved in error processing.

For this dissertation, a study was performed to understand which subroutine calls are avoided after conditional branches and which ones are actually taken after conditional branches. This was done to determine if there are any subroutine types that allow a branch prediction to be made on a conditional branch that guards the subroutine's execution. There did not appear to be any subroutine types that are useful when determining that a conditional branch will force the execution of a subroutine call. However, there were five subroutine types that were avoided, and that therefore allow a strong prediction to be made on the conditional branch guarding their execution. These subroutine types are as follows:

I/O Buffering – The subroutine calls of *filbuf*, used with the *getchar* macro, and *flsbuf*, used with the *putchar* macro, are avoided. These subroutines are only called when the I/O buffer is empty or full, and this is a rare occurrence. These subroutine calls allow the branch associated with the `?` operator in their corresponding macro to be predicted accurately.

Exiting – Any subroutine that forces the termination of program execution is avoided. The conditional exit of a program is usually only done when an exception has occurred. In addition, when an exit occurs, the program stops executing. Therefore, the compiler should assume that the program will continue running because this is the only case where the compiler can greatly improve the program's performance.

Error Processing – Error processing is a rare event, so any subroutine associated with error processing is avoided.

Memory Allocation – Many times the conditional execution of subroutines that allocate memory is avoided. While the application of the call heuristic in this case works well, the confidence the compiler can have in the heuristic being correct is less than in the previous three cases.

Printing – Printing that can be executed conditionally, even when it is not associated with `stderr`, is avoided. As in the case with the conditional execution of memory allocation, the confidence the compiler can have in the call heuristic being correct in this case is less than for the first three subroutine types described.

The enhancement made to the call heuristic in this dissertation is to understand which subroutine call types are forcing the application of the call heuristic. If the subroutine call is one of the five subroutine types described above, a strong assertion about the predicted branch direction can be made. The determination of subroutine call types is accomplished by running a preprocessor on the source code to determine which functions force the unconditional execution of a subroutine call found in one of the five subroutine types. Only subroutines found in one of the lists are used in the stronger prediction of conditional branches.

The preprocessor starts off with a list of seed functions for each of the five subroutine types. If a function in the benchmark allows a subroutine located in one of the lists to be executed unconditionally, that function is added to the same list. The preprocessor runs iteratively until all the subroutines that force an unconditional execution of one of the seed functions have been identified. The seed functions used in this dissertation are shown in Table 4.2.

Error processing is broken down into two parts in the seed functions so that some special situations can be taken into account. These special situations occur when there are conditions guarding the execution of the functions contained in the *error processing1* list, or error pro-

Table 4.2 Seed functions used to determine the special subroutines that are used with the call heuristic.

Subroutine Type	Subroutine Names
I/O Buffering	filbuf flsbuf
Exiting	exit abort
Error Processing1	yyerror error perror fprintf (to stderr) vfprintf (to stderr)
Error Processing2[a]	exit abort fatal warning
Memory Allocation	malloc calloc realloc
Printing	printf vprintf fprintf vfprintf fflush

[a] A "hit" is registered when part of a subroutine's name matches a string in this list.

cessing is done through error handlers that are called indirectly. Each of these cases does not allow the unconditional execution of a seed function to be determined, and would normally keep the special handling of these functions from being used in the call heuristic. Examining the subroutine names and determining if one of the strings in the *error processing2* list is present in a subroutine name allows these functions to be used in the call heuristic. While this pattern matching can lead to spurious functions being considered error processing functions, this problem was not detected in the benchmarks examined during this work. In fact, this functionality made a big difference in the *li* benchmark where there are a significant number of important

branches associated with error processing that fall into each of the special situations described earlier in this paragraph.

4.2.2.3 Loop header

> When a branch is derived from a conditional loop expression, predict that the successor that is a loop header or a loop preheader and does not postdominate is taken.

The Ball and Larus loop header heuristic is the same as the loop header heuristic used in this dissertation, except that it is applied to all branches guarding the execution of a loop. There is no check to ensure that a branch is derived from a conditional loop expression. As shown in Figure 4.2, `for` and `while` loops in the source code get transformed into an `if` statement, which contains a copy of the original conditional loop expression, guarding the execution of a `do-while` loop, which is a transformed version of the original loop.

The Ball and Larus loop header heuristic gets applied to either a branch that originally preceded the loop or a branch in an `if` statement that is derived from the original conditional loop expression. The different cases covered by the Ball and Larus loop header heuristic are shown in Figure 4.3. This figure has three different cases that are mutually exclusive, and cover all the cases caught by the original Ball and Larus loop header heuristic. The branches that have the Ball and Larus loop header heuristic applied to them are in bold in the figure. These cases are broken out in this dissertation so that better branch predictions can be made, and so that stronger assertions can be made about the predicted branch directions.

Figure 4.3(a) shows the case where the `for` loop initialization is variable. In this case, the derived `if` statement, "`if (init < loop_bound)`," cannot be optimized away. Figure 4.3(b) is similar to case (a) because the derived `if` statement, "`if (C1 < loop_bound)`," is not optimized away. The difference is that the `for` loop initialization is a constant. Case (b) is

```
if (j) {
    i = init;
    if (init < loop_bound)
        do {
            do_something;
            i++;
        } while (i < loop_bound);
}
```

(a)

```
if (j) {
    i = C1;
    if (C1 < loop_bound)
        do {
            do_something;
            i++;
        } while (i < loop_bound);
}
```

(b)

```
if (j) {
    i = C1;
    do {
        do_something;
        i++;
    } while (i < C2);
}
```

(c)

Figure 4.3 Different situations covered by Ball and Larus loop header heuristic: (a) variable initialization, (b) constant initialization and variable loop bound, (c) constant initialization and constant loop bound.

distinguished from case (a) because case (b) can have a stronger assertion made about the loop header branch. Since the initialization is constant, the loop is more likely to be executed at least once.

Figure 4.3(c) is different from the previous two cases because the derived `if` statement can be optimized away. In this case, another `if` statement, "`if (j)`," located before the original loop in the source code, has the Ball and Larus loop header heuristic applied to it. However, this branch does not have to be related to the loop at all, and it may not be biased so that the loop is entered.

Understanding these cases also helps explain a phenomenon described in a paper by Calder et al. [32]. In Calder's paper, it is noted that the Ball and Larus loop header heuristic performs well for C programs, but poorly for FORTRAN programs. The three cases covered here help

explain this phenomenon. FORTRAN programs contain a much higher percentage of loops that would fall into case (c). Because of this, the Ball and Larus loop header heuristic is not expected to function well on FORTRAN programs.

In this dissertation, only cases (a) and (b) have the loop header heuristic applied to them. In addition, cases (a) and (b) are distinguished because case (b) has a better chance of being correct. This is useful when determining how aggressive a compiler optimization can be with code located near a branch where the loop header heuristic is applied.

4.2.2.4 Return

Predict that the successor that contains a return is not taken.

The Ball and Larus return heuristic is identical to the return heuristic used in this dissertation. This heuristic seems to work well because of the way that most programmers use return statements. Returns are often used to exit a function early when an error or boundary case is detected. In these cases, the return statement is not likely to be taken.

4.2.2.5 Pointer

If a branch compares a pointer to NULL or compares two pointers, predict that the pointers are not equal as long as these pointers are not part of an array.

The Ball and Larus pointer heuristic is the same as the pointer heuristic used in this dissertation, except that a pointer operand that is part of an array does not get the pointer heuristic applied to its branch in this dissertation. This exception is made because an element in an array of pointers has a higher likelihood of being NULL than a pointer that is not part of an array. This is definitely true if the array of pointers is sparsely populated and if the array contains only a few non-NULL members.

An advantage that the pointer heuristic used in this dissertation has over most previous branch prediction work is that source-level information is available when applying the heuristic. Ball and Larus, along with Calder et al., used binary instrumentation to gather statistics and applied the branch prediction heuristics to these program traces. Because of this, they did not have access to source-code information to know which branch operands were pointers. Therefore, they looked for code patterns to determine whether branch operands were pointers, and these patterns may have captured some spurious cases that were not associated with a pointer comparison. The source-level information used in this dissertation allows the pointer heuristic to perform better.

The pointer heuristic works because pointers do not usually equal NULL or one another. While this is definitely not always the case, it seems to be true more often than it is false.

4.2.2.6 Opcode

> Predict that a comparison of an integer less than zero, less than or equal to zero, or two floating point values being equal fails. Predict that a comparison of an integer greater than zero or greater than or equal to zero succeeds.

The Ball and Larus opcode heuristic is identical to the opcode heuristic used in this dissertation. This heuristic seems to work well because integers normally contain positive numbers. Many times this heuristic can be applied simultaneously with the loop header heuristic, return heuristic, and call heuristic, when an error processing function is detected at one of the targets. The overlap with the loop header heuristic occurs in counting loops that decrement. In this situation, a loop header branch that does a comparison against zero is usually present. The return heuristic and call heuristic overlap occurs when negative numbers are used to denote error conditions.

4.2.2.7 Character comparison

If a branch is an integer comparison against a character constant, predict that the comparison fails.

The character comparison heuristic is a new heuristic defined in this dissertation that is based on the fact that ASCII characters are usually not equal to one particular constant character value. For example, in English text, the most common character is a space, and it occurs less than half of the time.

The heuristic is applied to *branch if equals* (`BEQ`) and *branch if not equals* (`BNE`) branches that contain one operand that is a character constant. Unfortunately, the EDG front end used in IMPACT converts all character constants to integer constants. This makes it impossible to distinguish constant character comparisons and regular constant integer or enumerated type comparisons. The integer constants checked are -1 (EOF), 9 ('\t'), 10 ('\n'), and 32 (' ') thru 126 ('~'). This implementation ends up capturing some extraneous cases that are not involved with characters, but that share constant values that overlap with these common ASCII character values. The check against –1 does well even when a program is not working with ASCII characters because it is a common method of marking an error.

4.2.2.8 Restricted opcode AND

If a comparison is made against one bit in a bit field, predict that the bit is not set.

The restricted opcode AND heuristic is another new heuristic defined in this dissertation that works by assuming that one particular bit in a bit field is normally not set. The intuition is that many times when bit fields are used, they are set in the minority case. In this way, the bit in that field is normally not set. This heuristic is one of the weakest heuristics used, but it does seem to work reasonably well.

The heuristic looks for a `BEQ` or `BNE` branch that contains a constant zero as one of its operands. For the heuristic to apply, the other operand must have resulted from a preceding AND operation, and one operand of that AND must be a constant, power-of-two value.

4.2.2.9 Other Ball and Larus heuristics

Two heuristics described by Ball and Larus are not used in the branch prediction heuristics of this dissertation. These heuristics are the guard and store heuristics. These heuristics are not used because an analysis of the cases where these heuristics applied did not reveal any intuition that explained the success of the heuristics.

The Ball and Larus guard heuristic states that if one of the branch's operands is used in a branch's target, assume the target containing the operand is taken. This heuristic assumes that a condition is being tested for, and the normal case allows the operations using the operand to execute. The most common situation for this occurs with a pointer check against NULL. Only if the pointer is non-NULL, is a load from the pointer allowed. However, this case is already captured by the pointer heuristic. Since the most common case for this heuristic ends up being a subset of the pointer heuristic, this heuristic is not applied in this dissertation.

The Ball and Larus store heuristic states that if one of the branch's targets contains a store, assume that the other target is taken. No intuition has been provided on why this heuristic works, and no potential intuitive reason for the success was detected when examining the cases where this heuristic performed well. Calder et al. found that this heuristic was wrong more than it was correct for the C programs evaluated in their paper [32].

4.2.2.10 Switch statements

> Predict that all the non-error processing targets of an indirect jump or branches associated with a `switch` statement are equally likely.

The branches associated with `switch` statements, and the corresponding indirect jump if the `switch` statement is converted into a multiway branch, are difficult to predict because useful work is performed for only one target of the branch. The other branch target is a branch for another `case` statement. This limits the amount of instructions found in both branch targets, and the branch prediction heuristics do not have the context for these situations that they do for branches associated with conditional expressions in `if` statements and loops. Since the context is limited, the heuristics used in this dissertation assume all the branches associated with a `switch` statement are equally likely. This means that branches associated with `switch` statements are assumed to go in the direction that allows the branch for the next `case` statement to execute.

The one exception to this occurs when an error processing or exiting subroutine call is present in a `case` statement target. In this situation, the target with the error processing or exiting subroutine call is assumed to never execute, and the corresponding branch for the `case` statement is predictable. As with these same situations in the call heuristic, these subroutine calls are almost always avoided.

4.2.2.11 Application order

As is the case with the Ball and Larus branch prediction heuristics, the heuristics in this dissertation are applied in a predetermined order. Once a branch prediction heuristic applies, no other lower-priority branch prediction heuristics are examined. First, all the loop branches are predicted as described in Section 4.2.2.1. Second, all branches associated with `switch` statements are predicted as described in Section 4.2.2.10. Finally, all the non-loop, non-switch branches are predicted in the order presented in Table 4.3.

Table 4.3 Branch prediction order for non-loop, non-switch branches.

Application Order
call associated with exiting
call associated with error processing1
call associated with error processing2
call associated with buffering
loop header with constant initialization and variable loop bound
call associated with printing
call associated with memory allocation
character comparison
return
loop header with variable initialization
pointer
opcode
call
restricted opcode AND

The application order was chosen based on the performance of the individual heuristics (e.g., an individual heuristic that is never mispredicted should be one of the first heuristics applied), and a pair-wise comparison of the heuristics when their predictions conflicted. As seen in the results presented in Section 4.2.3.6, the heuristics and application order used in this dissertation perform favorably when compared to the heuristics and application order used by Ball and Larus.

4.2.3 Results

This section presents the results of static branch prediction. The results for the heuristics used in this dissertation, as well as the results of an implementation of the heuristics described by Ball and Larus, are shown. In addition, in-depth analysis of several heuristics are shown in order to provide insights into the decisions made when defining the heuristics used in this dissertation. All of these heuristics are implemented in the IMPACT compiler.

Both sets of static branch prediction heuristics perform their prediction on the same Lcode files. The Lcode files that are used for prediction have gone through a minimal compilation path. The C source code is read in by IMPACT, flattened in Pcode, translated to Lcode, and optimized using IMPACT's classical optimizations. After optimization, the code is profiled. The results of profiling do not affect the actual static branch predictions, but profiling does allow the success of the branch prediction to be easily measured.

The branches that are predicted can be broken down into three categories: loop branches, switch branches, and non-loop, non-switch branches. Section 4.2.2.1 defines loop branches. The Ball and Larus heuristics use the strict definition of loop back edges and loop exits defined in that section, and the heuristics from this dissertation do not count loop back edges that have no effect on the loop trip count, as illustrated by the *grep* example in Section 4.2.2.1. Switch branches are branches that are derived from `switch` statements in the source code. During the minimal compilation path used when generating these results, no indirect jumps are used, so every `switch` statement is converted into a set of adjacent branches. The remaining branches are non-loop, non-switch branches.

Non-loop, non-switch branches are the interesting branches to predict. As mentioned in the previous section, loop branches are predicted so that loops continue to iterate, and switch branches are predicted so that every target of the `switch` statement is equally likely. The remaining branches are difficult branches to predict, and represent a large portion of the branches that are dynamically executed. The results in this section focus on these non-loop, non-switch branches.

The benchmarks used in this analysis are the same ones presented in Chapter 3. Even though the results are presented for these benchmarks, most of the analysis performed in devel-

oping the static branch prediction heuristics was performed on a different set of benchmarks. These benchmarks included nine Unix utilities (*cccp*, *cmp*, *eqn*, *grep*, *lex*, *qsort*, *tbl*, *wc*, and *yacc*), the six benchmarks present in SPEC CINT92, and *alvinn* and *ear* from SPEC CFP92. These benchmarks represent the training set used to understand and evolve the static branch prediction heuristics. There is some overlap in the benchmarks used during training and the benchmarks used for generating the results, but a majority of the benchmarks used for the results were not present in the training set. A training set was purposely used when developing these heuristics in order to not develop a set of "tuned" heuristics that perform well for the benchmarks that are analyzed, but do not perform well for other programs.

The remainder of this section consists of the following subsections. Section 4.2.3.1 shows the results obtained by the Ball and Larus heuristics and the dissertation heuristics for all the non-loop, non-switch branches. In-depth analyses of individual heuristics are found in Sections 4.2.3.2, 4.2.3.3, 4.2.3.4, and 4.2.3.5. Finally, Section 4.2.3.6 compares the effectiveness of the Ball and Larus heuristics and the dissertation heuristics.

4.2.3.1 Non-loop branch prediction

As mentioned earlier, the Ball and Larus heuristics described in the paper, "Branch Prediction for Free" [26], are implemented in the IMPACT compiler. The effectiveness of each Ball and Larus heuristic, applied individually, for predicting non-loop, non-switch branches is shown in Table 4.4. These results are independent of the heuristic combining that is needed to make a prediction for individual branches. This means that a branch can be counted in multiple heuristic columns if the conditions for different heuristics apply simultaneously.

Table 4.4 Branch prediction results for Ball and Larus heuristics.

Benchmark - Input Pair	NL %	Call		Loop Header		Pointer		Return		Opcode		Guard		Store	
espresso - 1	36	2	5/66	21	3/80	3	2/99	4	30/95			27	8/91	21	21/82
espresso - 2	59	3	7/54	13	7/75	9	1/99	2	12/72	2	2/99	26	9/97	43	17/89
espresso - 3	57	6	8/58	22	8/57	9	6/95	4	9/82	2	5/96	29	7/91	32	24/75
sc - 1	54	20	11/89	8	1/92	26	29/63	7	2/83	4	3/69	36	20/70	11	4/40
sc - 2	55	23	8/92	9	0/96	38	22/63	4	3/92	2	5/60	42	19/84	8	4/40
sc - 3	45	19	10/80	7	0/93	33	26/92	14	0/87	8	0/65	44	18/76	11	1/65
go - 1	68	15	20/56	16	18/95			17	19/96	2	23/93	37	20/86	31	20/58
go - 2	68	16	20/57	16	19/96			18	20/96	2	23/90	37	20/86	32	20/59
go - 3	70	16	20/58	16	18/97			18	20/97	2	23/88	37	22/84	30	21/60
m88ksim - 1	52	13	9/69	3	7/90	4	4/92	14	6/66			19	8/77	11	10/56
m88ksim - 2	52	12	6/68	3	1/99	4	0/100	14	4/65			20	7/78	10	9/55
m88ksim - 3	55	12	4/77	6	1/83	12	2/95	12	3/76			18	6/83	8	8/60
gcc - 1	38	22	11/68	13	10/60	17	13/65	17	11/75	7	11/78	39	12/67	20	10/59
gcc - 2	43	20	12/64	11	11/62	19	13/68	19	10/78	6	8/79	39	12/68	17	10/61
gcc - 3	30	19	10/75	8	6/72	10	10/64	12	11/73	7	7/79	37	9/72	26	7/73
compress - 1	55	25	25/63	4	10/26			5	0/34	44	14/56	42	15/93	36	6/27
compress - 2	24	28	23/24	6	10/21			7	2/33	31	0/52	46	12/98	45	3/29
compress - 3	64	28	23/42	6	10/22			7	0/32	31	2/55	48	13/96	44	7/24
li - 1	43	41	5/79	27	4/82	55	2/90	40	4/77			31	1/91	34	14/50
li - 2	33	40	10/76	36	16/78	52	8/92	35	6/80			33	10/94	41	16/41
li - 3	52	42	8/86	36	8/94	58	3/93	43	7/83			35	4/94	38	15/69
ijpeg - 1	50	15	6/92	10	18/30			4	22/51	22	43/99	31	12/32	11	3/54
ijpeg - 2	51	15	7/94	11	19/36			5	22/54	19	42/100	33	12/30	15	3/66
ijpeg - 3	72	22	4/100	14	17/41			4	27/24	17	19/100	33	15/76	24	5/68
perl - 1	31	27	2/90	2	32/8	23	0/65	19	0/59	1	0/100	34	8/41	29	11/24
perl - 2	26	38	3/74	11	1/96	42	2/86	30	2/80	2	1/62	43	5/64	18	10/33
perl - 3	30	30	2/77	3	2/93	38	7/79	29	2/75	2	0/100	50	8/60	21	11/23
vortex - 1	83	40	0/79			4	0/96	2	0/64			29	1/25	40	0/70
vortex - 2	93	43	0/79			4	0/97	3	0/61			31	1/25	37	0/65
vortex - 3	96	44	0/80			4	0/96	3	0/62			32	1/25	36	0/63
Pcode - 1	53	29	5/80	7	15/78	35	14/73	17	12/67	4	18/92	46	9/80	23	7/66
Pcode - 2	50	30	4/84	6	13/89	26	18/81	16	9/63	5	28/94	39	10/81	23	7/50
Pcode - 3	48	31	9/78	10	8/15	13	4/60	7	4/48	19	27/22	28	3/24	6	6/34
Lhppa - 1	44	50	8/87	11	6/90	53	10/80	32	5/75	6	3/90	34	11/66	20	5/46
Lhppa - 2	68	46	8/91	14	4/48	56	14/62	25	5/82	14	1/98	46	13/54	23	3/42
Lhppa - 3	37	41	8/82	12	12/85	41	9/82	24	5/78	4	4/90	40	9/76	25	5/24

The predictions are performed on the benchmarks and inputs described in Chapter 3. The results are based on the dynamic behavior observed when the code is actually executed. So a branch gets counted every time that it is executed during a program run. For each benchmark-input pair, the percentage of non-loop, non-switch branches are present in the *NL %* column. This represents the percentage of the total branches executed during a program run that can

be predicted by the heuristics present in this table. The results of each heuristic are shown with three numbers. The number on the left of each column represents the percentage of non-loop, non-switch branches executed dynamically that have the heuristic applied to them. If a heuristic is applied to less than 1% of the non-loop, non-switch branches, the entire column entry is blank. The two numbers on the right of each column represent the effectiveness of the predictions. The number to the left of the slash contains the miss rate for a perfect predictor, and the number to the right of the slash contains the percentage of branches in which the heuristic made the correct prediction. A correct prediction means the heuristic made the same decision that a perfect predictor would make.

The heuristics developed in this dissertation are also implemented in the IMPACT compiler. The effectiveness of these heuristics, applied individually, is shown in Table 4.5. The meaning of the numbers is this table match the meaning of the numbers in Table 4.4.

The results of the two prediction tables are now compared. The percentage of non-loop, non-switch branches evaluated by the two sets of heuristics differ by a small amount in a few of the benchmark-input pairs. This difference, described in Section 4.2.2.1, occurs because Ball and Larus use a strict definition for a loop back-edge branch. The heuristics used in this dissertation detect conditions where a loop back-edge branch does not affect the loop iteration; therefore, these branches are not considered loop back-edge branches by the heuristics. This is not a common situation, but it is detected in *sc-1*, *perl-2*, and all the *li* cases. The *li* cases are affected the most. For *li*'s inputs 1, 2, and 3, the dissertation heuristics consider 2.8%, 5.5%, and 2.1%, respectively, of the dynamically executed branches as non-loop, non-switch branches, while the Ball and Larus heuristics consider them loop branches. All these cases occur in loops

Table 4.5 Branch prediction results for heuristics used in this dissertation.

Benchmark - Input Pair	NL %	Call[a]	Loop Header	Pointer	Return	Opcode	Char. Comp.	Restr. Opc. AND
espresso - 1	36	1 3/99	17 0/100	3 2/99	4 30/95		14 0/100	39 18/79
espresso - 2	59	1 6/97	8 0/74	9 1/99	2 12/72	2 2/99	9 0/100	42 19/89
espresso - 3	57	3 10/95	12 1/74	8 4/95	4 9/82	2 5/96	8 0/100	40 25/92
sc - 1	55	2 5/96	8 1/92	5 32/32	7 2/83	4 3/69		7 5/70
sc - 2	55		9 0/96	10 10/8	4 3/92	2 5/60		11 22/100
sc - 3	45	2 1/98	7 0/93	6 27/84	14 0/87	8 0/65		8 1/90
go - 1	68		10 11/95		17 19/96	2 23/93	2 25/64	
go - 2	68		10 12/95		18 20/96	2 23/90	1 24/75	
go - 3	70		11 11/96		18 20/97	2 23/88	1 17/94	
m88ksim - 1	52		1 4/82	4 5/92	14 6/66		3 3/99	2 16/79
m88ksim - 2	52		1 2/100	3 0/100	14 4/65		3 4/100	
m88ksim - 3	55		4 2/88	9 3/93	12 3/76		2 2/99	2 2/79
gcc - 1	38	3 0/93	12 10/62	10 11/66	17 11/75	7 11/78	25 11/71	
gcc - 2	43	2 0/92	10 11/65	11 12/64	19 10/78	6 8/79	31 11/74	
gcc - 3	30	5 0/97	8 4/76	6 9/66	12 11/73	7 7/79	27 11/82	
compress - 1	55				5 0/34	44 14/56		
compress - 2	24				7 2/33	31 0/52		
compress - 3	64				7 0/32	31 2/55		
li - 1	46	18 0/100	21 0/90	32 3/85	37 4/77			11 37/70
li - 2	39	14 0/100	20 11/92	30 10/87	30 6/80			23 30/37
li - 3	54	15 0/100	21 0/92	37 4/90	41 7/83			13 31/6
ijpeg - 1	50		9 21/36		4 22/51	22 43/99		
ijpeg - 2	51		9 22/44		5 22/54	19 42/100		
ijpeg - 3	72	8 0/100	12 18/46		4 27/24	17 19/100	4 0/100	
perl - 1	31	3 0/100	2 32/8	17 0/54	19 0/59	1 0/100		15 4/76
perl - 2	27	12 1/97	11 1/96	35 2/86	30 2/80	2 1/62	1 4/62	7 12/68
perl - 3	30	10 2/99	3 2/93	33 8/76	28 2/75	2 0/100	2 0/100	6 8/91
vortex - 1	83			1 1/89	2 0/64			9 0/0
vortex - 2	93			2 1/91	3 0/61			10 0/0
vortex - 3	96			2 1/89	3 0/62			10 0/0
Pcode - 1	53	10 4/92	7 15/78	22 21/53	17 12/67	4 18/92	13 8/98	
Pcode - 2	50	18 2/97	6 11/89	16 27/66	16 9/63	5 28/94	17 5/98	
Pcode - 3	48	5 1/100	10 8/12	12 3/59	7 4/48	19 27/22		
Lhppa - 1	44	32 1/98	11 6/92	32 7/87	32 5/75	6 3/90	4 2/85	2 14/65
Lhppa - 2	68	21 1/99	13 4/44	20 7/90	25 5/82	14 1/98		2 4/92
Lhppa - 3	37	27 1/99	12 13/87	25 6/88	24 5/78	4 4/90	8 2/91	

[a]Only instances of the call heuristic associated with I/O buffering, exiting, error processing, memory allocation, and printing are counted here. For the results of all instances of the call heuristic, reference the *call* column of Table 4.4

where the conditional loop expression is always true. The Ball and Larus loop branch heuristic is only correct 45%, 46%, and 43% of the time for the branches in question for the three inputs.

The call heuristic is used in both sets of heuristics. In fact, the call heuristic is applied to the branches in a similar way for both prediction heuristics. The only difference is that the dissertation heuristics detect special cases. These special cases are described in Section 4.2.2.2 and allow a higher level of confidence to be present for branches that fall into one of these cases. The combined results for the special cases are shown in the *call* column of Table 4.5. The combined results for all applications of the call heuristic are shown in the *call* column of Table 4.4. As seen in Table 4.5, the special cases of the call heuristic do apply to a significant number of important branches, and these cases have a much higher bias than the non-special case instances of the call heuristic. A detailed analysis of the success of the call heuristic categories is present in Section 4.2.3.2.

The loop header heuristic is used in both sets of heuristics, but the version of the heuristic used in this dissertation does not apply to as many branches as the version used by Ball and Larus. A description of the differences is described in Section 4.2.2.3. Table 4.5 again combines the results of the two situations detected by the dissertation heuristics. A more detailed analysis of the loop header heuristic categories, and the effectiveness of splitting the heuristic into the different cases, is present in Section 4.2.3.3.

The pointer heuristic is used in both sets of heuristics, but the version of the heuristic used in this dissertation does not apply the heuristic when the pointer originated from an array. A detailed description of this modification is present in Section 4.2.2.5. A major difference in the Ball and Larus pointer heuristic described in their paper, and the implementation done in IMPACT, is the use of type information. This allows the Ball and Larus pointer heuristic to

be applied only to cases involving pointers, and not to any extraneous cases that could happen in the implementation described in the original Ball and Larus work. A more detailed analysis of the pointer heuristic results is present in Section 4.2.3.4.

The next two heuristics in both tables, return and opcode, are identical for both sets of heuristics. A detailed description of the return heuristic is present in Section 4.2.2.4, and a detailed description of the opcode heuristic is present in Section 4.2.2.6.

The remaining heuristics in both tables are implemented only in their respective sets of heuristics. The guard and store heuristics are used in the Ball and Larus heuristics. A description of why the guard and store heuristics are not used in the dissertation heuristics is present in Section 4.2.2.9.

The guard heuristic looks like it performs well when the results from Table 4.4 are examined. Seven out of the 36 cases predict a majority of the dynamic branches incorrectly, but 23 out of the 36 cases predict over 70% of the dynamic branches correctly. However, the analysis present in Section 4.2.3.5 shows that this is misleading because a majority of the winning cases are actually covered by the pointer heuristic.

The store heuristic does not do well. It predicts a majority of the dynamic branches incorrectly in 13 out of 36 cases. In addition, it only predicts over 70% of the cases correctly five times, and three of those cases come from the *espresso* benchmark. This poor performance actually matches results obtained by Calder et al. with their implementation of the Ball and Larus store heuristic [32].

The unique heuristics contained in the dissertation heuristics are the character comparison and restricted opcode AND heuristics. These new heuristics are described in Sections 4.2.2.7 and 4.2.2.8.

The character comparison heuristic performs well. For the 19 cases where at least 1% of the dynamic branches can have the character comparison applied to them, no case predicts a majority of the dynamic branches incorrectly. In addition, 17 out the 19 cases predict over 70% of the dynamic branches correctly. This heuristic is being applied to more cases than just character comparison. For example, the majority of dynamic branches in *espresso* and *ijpeg* are being correctly predicted by the character comparison because they use –1 to denote an infrequently occurring case. This allows the heuristic to show wins even when there is no character processing taking place.

The restricted opcode AND heuristic also performs well, but not as well as most of the other heuristics. It ends up mispredicting a majority of the dynamic branches for 4 out of 19 cases, and it correctly predicts at least 70% of the dynamic branches for 12 out of the 19 cases.

The results for the remaining sections will be presented differently than the way they were presented in this section. Results are shown in a graphical form, and the percentages of correct prediction are emphasized. The percentage of heuristic application is used as a filter so that only cases where more than 1% of the dynamic branches are involved get evaluated, but they are ignored otherwise. In addition, the miss rate of the perfect predictor is also ignored.

4.2.3.2 Call heuristic

The improvement to the call heuristic made in this dissertation is to classify situations where a branch direction decision is more biased. A description of these classifications is found in Section 4.2.2.2. The compiler can take advantage of these biased conditions by being more aggressive in the optimizations that it performs when the biased conditions are recognized.

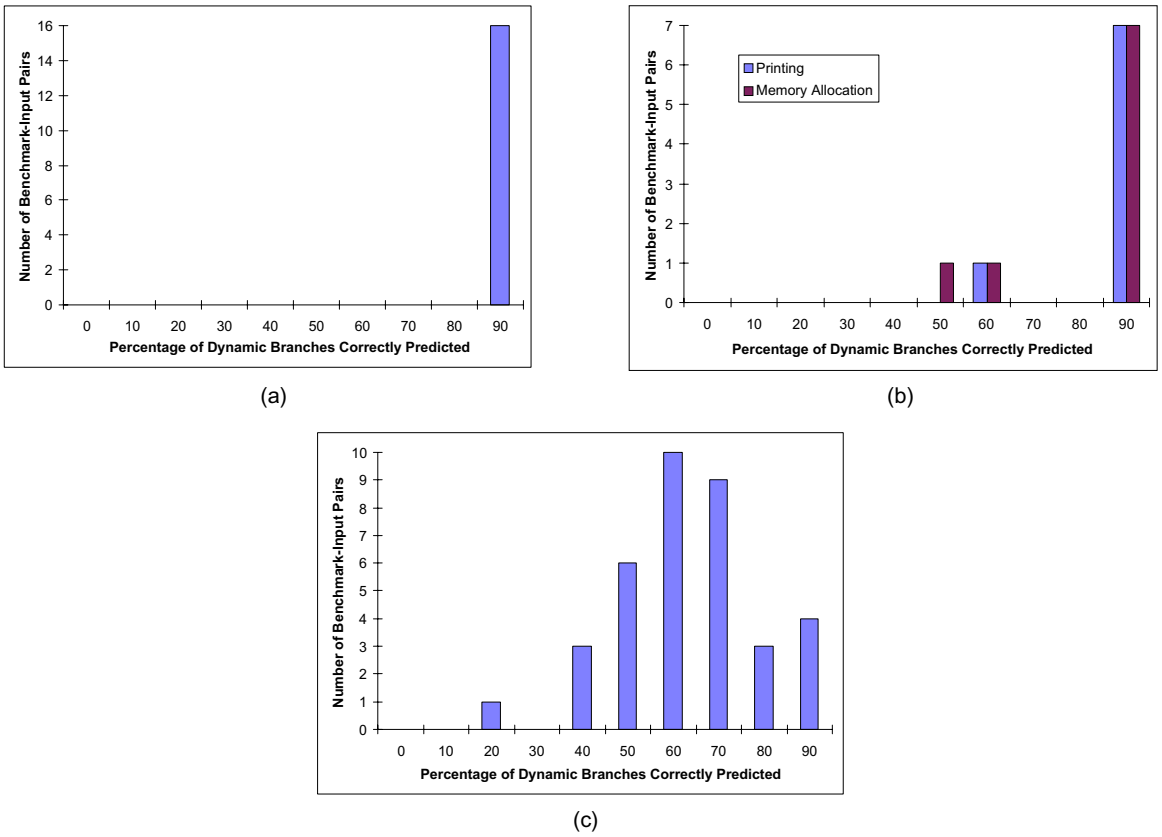

Figure 4.4 Success of different classifications of the call heuristic: (a) I/O buffering, exiting, and error processing, (b) memory allocation and printing, (c) all other cases.

The results for the different classifications are shown in Figure 4.4. The results in all three parts of the figure show the number of benchmark-input pairs that fit into each percentile of dynamic branches correctly predicted. A benchmark-input pair gets counted in this result if at least 1% of the dynamic non-loop, non-switch branches executed fit into the category of interest. Each benchmark has three different inputs applied to it, so one benchmark may be counted up to three times. The percentiles are broken into 10% increments, such that all benchmark-input pairs that are correct 90% of the time or more get put in the *90* percentile column, while pairs that are correct at least 80% of the time but less than 90% are in the *80* percentile column, and so on.

Figure 4.4(a) shows the combined results of the three most biased situations: I/O buffering, exiting, and error processing. The bias works two ways. First, almost all of the exiting and error processing calls located after a branch are never executed. In addition, the branches guarding the execution of I/O buffering only refill the buffer about once every 8000 executions. Second, the biased direction is always correctly predicted. It should be noted that for all the training and reference benchmarks used in the analysis, no spurious functions were considered part of the error processing2 condition described in Section 4.2.2.2.

Figure 4.4(b) shows the results of the other two biased situations: memory allocation and printing. The results of the individual heuristics are both presented in this graph. These two cases allow better predictions, but they are not as biased as the cases present in part (a) of the figure. The memory allocation cases have an average perfect miss rate of 4%, with a maximum perfect miss rate of 18% for *Pcode*-input2. The two benchmark-input pairs with a correctness percentage less than 90 are present in *Pcode*-input1 (64% of the dynamic branches are incorrectly predicted) and *Pcode*-input2 (54% of the dynamic branches are incorrectly predicted). The printing cases are highly biased, with a maximum perfect miss rate of 1.6%. The only benchmark-input pair with a correctness percentage less than 90 belongs to *Lhppa*-input1, which predicts 64% of the dynamic branches correctly.

Finally, Figure 4.4(c) shows the results of all other instances of the call heuristic. The average perfect miss rate for these cases is 11%, with a maximum perfect miss rate of 25%. The four benchmark-input pairs that predict less than 50% of the dynamic branches correctly are *espresso*-input2, *espresso*-input3, *compress*-input2, and *compress*-input3. One reason why the remaining call heuristics do so well is that not all of the functions belonging to one of the five special categories are identified by the callname preprocessor. The same reason that dictated the

use of error processing2 for the *li* benchmark, may be working for the other categories. There are other functions that are also performing error processing, memory allocation, printing, etc. that are not being identified as such by the callname preprocessor.

4.2.3.3 Loop header heuristic

The Ball and Larus loop header heuristic is improved upon in this dissertation by breaking up the situations where their heuristic applied into three distinct cases. The definition of each of these cases is described in Section 4.2.2.3. The results for the three different cases are presented in Figure 4.5. Again, the results are filtered so that only instances where at least 1% of the non-loop, non-switch branches are involved are present.

Figure 4.5(a) shows the results of the variable initialization part of the Ball and Larus loop header heuristic. In this case, the branch derived from the conditional loop expression is guarding a loop's execution. As seen in the figure, the majority of benchmark-input pairs predict the variable initialization case correctly. Of the losing cases, only *ijpeg* is always a loser. The benchmark *ijpeg* is correct for only 16%, 27%, and 18% of the dynamic branches for its inputs 1, 2, and 3 respectively. The other two losing benchmark-inputs pairs are *perl*-input1 and *Pcode*-input3, but all of the other inputs for these two benchmarks are correct for at least 75% of the branches in this case.

Figure 4.5(b) shows the results of the constant initialization, variable loop bound case of the Ball and Larus loop header heuristic. The branch guarding a loop's execution is again derived from a conditional loop expression. However, this case is more biased than the variable initialization case shown in Figure 4.5(a). There are only three losing benchmark-input pair cases, and no benchmark always loses. *Pcode*-input3 is correct for only 2% of the dynamic

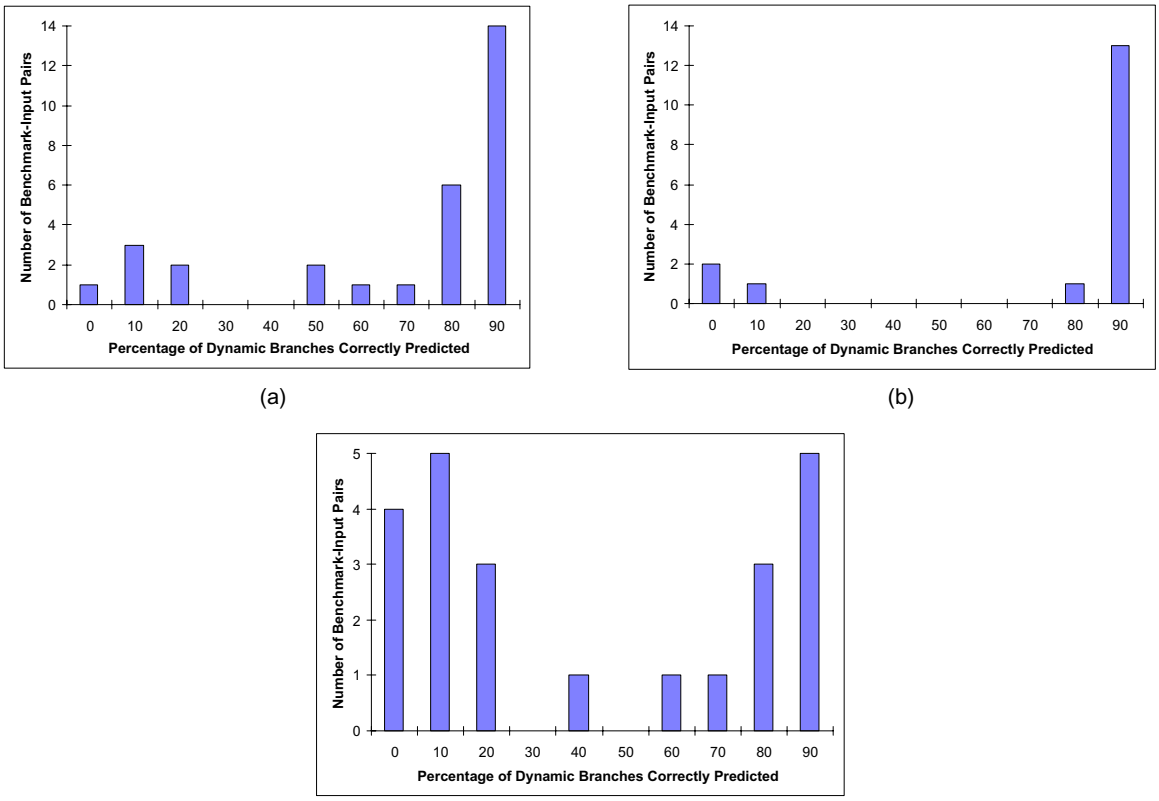

Figure 4.5 Success of different classifications of the loop header heuristic: (a) variable initialization, (b) constant initialization and variable loop bound, (c) constant initialization and constant loop bound.

branches, but the other two inputs of Pcode are correct for over 98% of the dynamic branches. The other two losing cases come from *espresso*'s inputs 2 and 3. There is a fundamental difference in the input data for *espresso*-input1 (input1 uses binary variables and inputs 2 and 3 do not), and all cases of constant initialization, variable loop bound are predicted correctly for this input. The perfect miss rates are also different between the Figure 4.5(a) and Figure 4.5(b) cases. The Figure 4.5(a) cases have an average perfect miss rate of 8.4% with a maximum perfect miss rate of 32.4%; the Figure 4.5(b) cases have an average perfect miss rate of 4.5% with a maximum perfect miss rate of 15.7%.

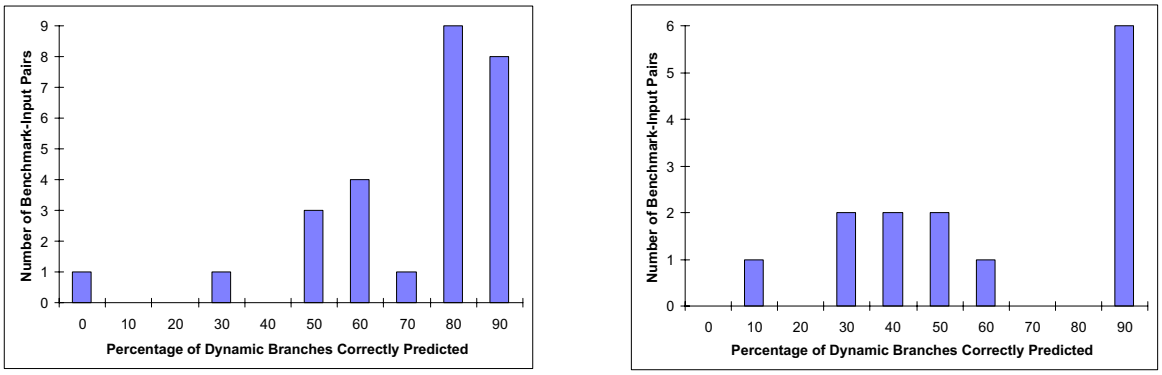

Figure 4.6 Success of different classifications of the pointer heuristic: (a) pointers are not located in arrays, (b) pointers are located in arrays.

Finally, Figure 4.5(c) shows the results of constant initialization and constant loop bound. For this case, the branch guarding the loop's execution has nothing to do with conditional loop expression. This case is captured by the Ball and Larus heuristic, but is ignored by the heuristics used in this dissertation. As seen in the figure, cases where more dynamic branches are mispredicted outnumber cases where correct predictions dominate. This enhancement is able to help the *compress* benchmark by filtering out the poor prediction percentages of 26%, 21%, and 22% for its three inputs.

4.2.3.4 Pointer heuristic

The Ball and Larus pointer heuristic is improved in this dissertation by ignoring cases where the pointer is derived from an array. A detailed description of this improvement is present in Section 4.2.2.5. Figure 4.6 shows the two cases that cover the Ball and Larus heuristics: pointers not derived from arrays and pointers derived from arrays. As defined for the other figures in the results section, only cases accounting for 1% of the dynamic non-loop, non-switch branches are shown.

Figure 4.6(a) shows the case where the pointer is not derived from an array. As seen in the figure, there are only two benchmark-input pairs that mispredict a majority of the dynamic branches fitting this category. Both of these cases are from *sc*'s inputs 1 and 2. The other *sc* input, input 3, actually predicts 84% of the dynamic branches correctly.

Figure 4.6(b) shows the case where the pointer is derived from an array. This case is filtered out by the heuristics used in this dissertation, but is included in Ball and Larus's pointer heuristic. As seen in the figure, this situation is not correct as often as the Figure 4.6(a) case. The (b) case is correct more than it is wrong for the reference benchmarks, but this is not the case for the benchmarks only contained in the training set. For the training-set-only benchmarks, seven benchmark-input pairs are mispredicted for the majority of the dynamic branches, and no benchmark-input pairs are correctly predicted for the majority of the dynamic branches.

4.2.3.5 Guard heuristic

The Ball and Larus guard heuristic is not used in the heuristics used in this dissertation because a large number of cases caught by the guard heuristic are actually a subset of the pointer heuristic. More information on the decision behind not using the guard heuristic is present in Section 4.2.2.9. The results comparing the situations when the Ball and Larus guard and pointer heuristics apply are shown in Figure 4.7.

Figure 4.7(a) shows the case where the Ball and Larus guard and pointer heuristics apply simultaneously. As seen in the figure, this case is almost always predicted correctly for the reference benchmarks used in this analysis. One of the benchmarks used for training, *eqn*, is an example of a program where this case is always mispredicted.

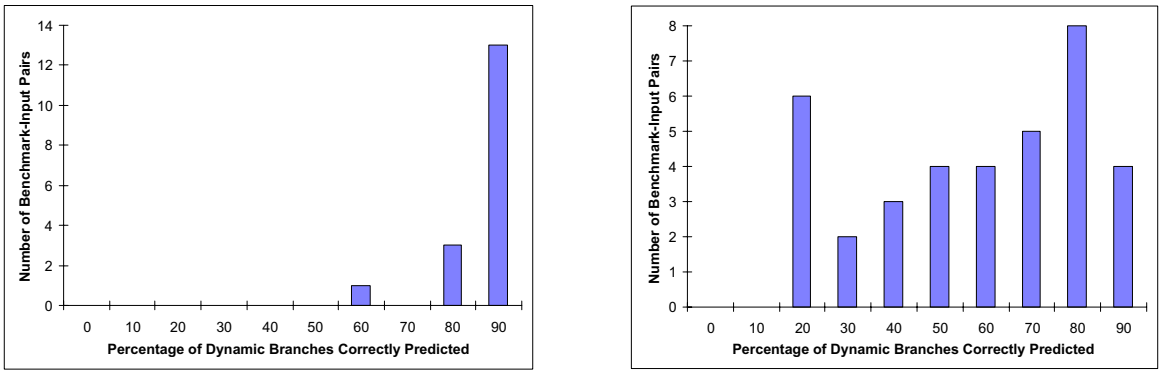

Figure 4.7 Success of different classifications of the Ball and Larus guard heuristic: (a) when the pointer heuristic applies, (b) all other cases.

Figure 4.7(b) shows the results for all the dynamic branches that are covered by the Ball and Larus guard heuristic, but not the pointer heuristic. This situation is much less biased, but still is correct for a majority of the dynamic branches a little over twice as much as it is incorrect. The training-set-only benchmarks are less biased. For the training-set-only benchmarks, 14 predict a majority of the dynamic branches correctly, and 13 predict a majority of the dynamic branches incorrectly.

4.2.3.6 Branch prediction decisions

The previous results evaluated the non-loop, non-switch branch heuristics individually, and did not analyze the effectiveness of the predictions made by the complete set of heuristics. Both the Ball and Larus heuristic set and this dissertation's heuristic set use a predefined application order of the heuristics to make an actual prediction for a branch. The Ball and Larus application order is discussed in Section 4.2.1, and the application order for the heuristics in this dissertation is discussed in Section 4.2.2.11. This section evaluates the effectiveness of the Ball and Larus heuristic set and this dissertation's heuristic set in predicting the direction of branches.

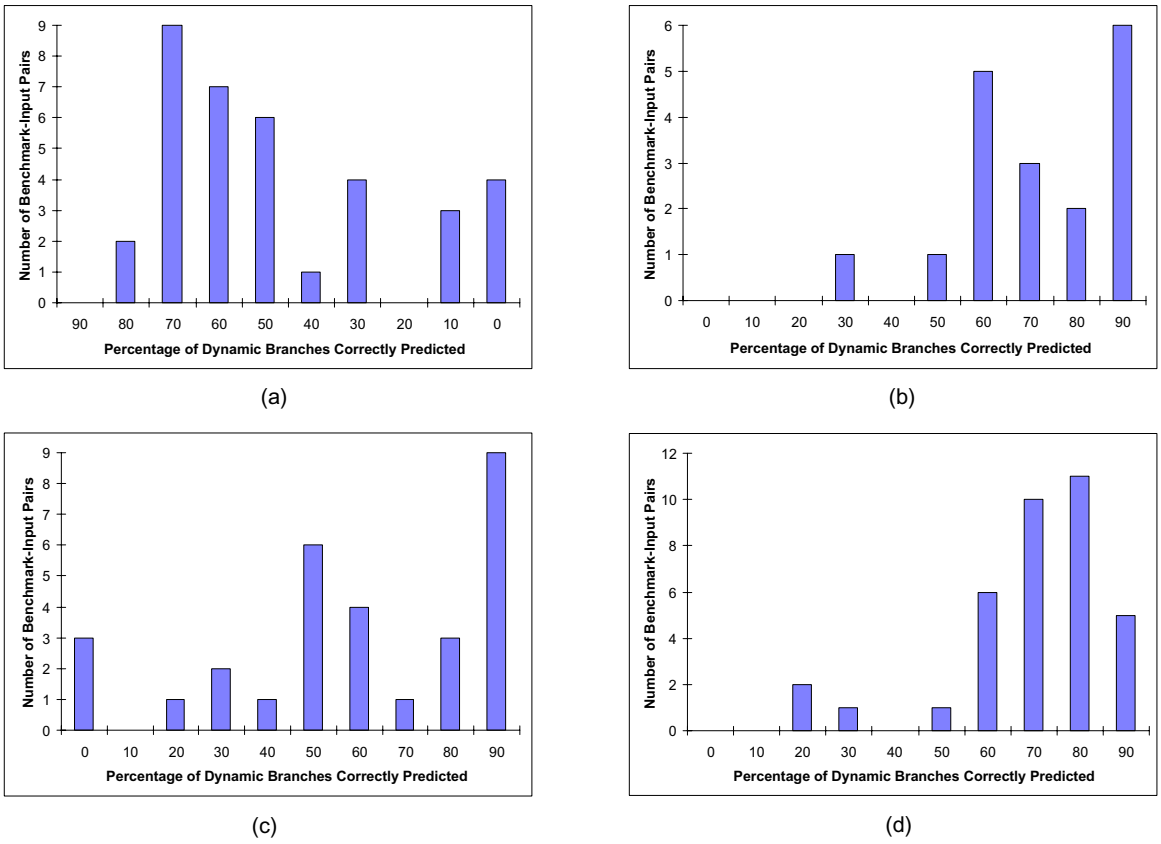

Figure 4.8 Comparison of heuristic predictions used in this dissertation to Ball and Larus heuristic predictions: (a) only Ball and Larus heuristics predict, (b) only heuristics used in this dissertation predict, (c) heuristic predictions from this dissertation disagree with Ball and Larus heuristic predictions, (d) heuristic predictions of Ball and Larus and the heuristic predictions used in this dissertation agree.

Figure 4.8 includes the results for four cases. These cases include one set of heuristics making a branch prediction decision and the other set of heuristics making no prediction, the two sets of heuristics making different branch prediction decisions, and the two sets of heuristics making the same branch prediction decision.

Figure 4.8(a) shows the results when Ball and Larus heuristics make a prediction, and the heuristics from this dissertation do not make a prediction for the same dynamic branches. The x-axis is swapped in this graph so that the right side of the graph represents cases in which the dissertation heuristics perform better. For consistency, this is done in all the graphs of

the figure. The right-hand side of each graph represents "winning" cases for the dissertation heuristics.

Differences can occur when one of the heuristics that is not used in this dissertation (e.g., the store or guard heuristic) or one of the cases that is filtered out by the dissertation heuristics (e.g., loop header header when the branch is not derived from a conditional loop expression) yields the prediction for the Ball and Larus heuristics. As seen in the figure, the Ball and Larus heuristics actually perform fairly well for the reference benchmarks (24 benchmark-input pairs have the Ball and Larus making the correct prediction for the majority of the dynamic branches, and 12 benchmark-input pairs have predictions that are incorrect for the majority of the branches). The training-set-only benchmarks do not perform as well (13 benchmark-input pairs have predictions that are correct for the majority of the dynamic branches, and 8 benchmark-input pairs have predictions that are incorrect for the majority of the branches).

Figure 4.8(b) shows the results when the dissertation heuristics make a prediction, and the Ball and Larus heuristics do not make a prediction for the same dynamic branches. This can occur when one of the new heuristics in this dissertation (e.g., character comparison or restricted opcode AND heuristic) is used to predict a branch direction. The only benchmark-input pair that loses in this case is *sc*'s input 1, but the other two inputs for *sc* are correct at least 80% of the time. The training-set-only benchmarks are just as biased, with seven benchmark-input pairs being predicted correctly and zero benchmark-input pairs being predicted incorrectly.

Figure 4.8(c) shows the results when the Ball and Larus heuristic set and the dissertation's heuristic set disagree on the direction of a branch. This can occur because of the differing heuristics and application order of the two heuristic sets. As seen in the figure, the dissertation's

heuristic set performs better than the Ball and Larus heuristic set. A similar trend is seen for the training-set-only benchmarks used in this analysis.

Figure 4.8(d) shows the results when both sets of heuristics make the same prediction for a branch. As seen in the figure, the dynamic branches are predicted correctly most of the time. When both heuristics make the same prediction, the only case where a majority of the dynamic branches are mispredicted occurs for all three inputs of *compress*. The training-set-only benchmarks are also predicted correctly almost all of the time for this case.

The bottom-line prediction success of the Ball and Larus heuristics and the heuristics used in this dissertation is shown in Table 4.6. The *miss rate* for a perfect predictor and the percentage of the dynamic branches predicted correctly, *correct %*, are presented for the cases when predictions are made. For each set of heuristics, the results are shown when all the branches are considered and when only the non-loop, non-switch branches are considered. For fairness, the Ball and Larus heuristics use the same predictions for branches associated with `switch` statements as the heuristics used in this dissertation.

As seen in the table, the heuristics used in this dissertation are able to produce better predictions than the Ball and Larus heuristics. For 10 out of the 12 benchmarks, the heuristics used in this dissertation provide higher percentages of correct predictions than the Ball and Larus heuristics. In three of these case, the heuristics used in this dissertation have percentages of correct predictions for the non-loop, non-switch branches that exceed the Ball and Larus correct percentages by 10%. The benchmarks *compress* and *vortex* are the only benchmarks in which the the Ball and Larus heuristics predict a higher percentage of correct branches for every input.

Table 4.6 Branch prediction success for both sets of heuristics.

	Ball and Larus				Dissertation			
	All Branches		Non-Loop Branches		All Branches		Non-Loop Branches	
Benchmark - Input Pair	Miss Rate	Correct %	Miss Rate	Correct %	Miss Rate	Correct %	Miss Rate	Correct %
espresso - 1	12	90	12	84	12	89	11	81
espresso - 2	16	78	13	87	16	79	12	88
espresso - 3	12	76	14	76	12	81	15	87
sc - 1	8	71	16	44	6	81	11	67
sc - 2	10	79	15	72	8	81	11	76
sc - 3	8	82	15	64	5	92	7	95
go - 1	22	77	20	70	22	84	19	80
go - 2	22	77	21	71	23	84	20	81
go - 3	24	75	21	69	24	84	19	83
m88ksim - 1	6	87	8	66	5	91	7	72
m88ksim - 2	5	88	6	67	5	92	5	73
m88ksim - 3	5	86	4	79	5	88	2	84
gcc - 1	7	84	11	64	7	86	11	68
gcc - 2	8	82	11	63	8	83	12	67
gcc - 3	5	91	9	73	5	92	9	76
compress - 1	15	73	17	45	17	72	21	29
compress - 2	4	90	11	48	4	91	15	26
compress - 3	14	69	13	55	15	65	15	34
li - 1	11	80	8	76	11	82	8	82
li - 2	11	76	14	64	11	78	15	69
li - 3	10	72	9	85	11	74	9	89
ijpeg - 1	11	82	21	60	11	92	28	83
ijpeg - 2	10	81	18	59	11	92	27	81
ijpeg - 3	13	84	13	80	13	84	13	80
perl - 1	4	86	4	59	4	89	2	65
perl - 2	5	89	4	70	4	90	3	73
perl - 3	4	90	7	67	4	93	6	75
vortex - 1	1	79	0	74	0	75	0	67
vortex - 2	1	73	1	71	1	70	0	67
vortex - 3	1	71	1	70	1	68	0	67
Pcode - 1	10	81	11	75	9	83	10	79
Pcode - 2	10	84	13	80	9	86	10	85
Pcode - 3	7	77	9	48	7	79	12	46
Lhppa - 1	8	87	8	78	8	89	8	83
Lhppa - 2	10	76	10	71	9	81	8	78
Lhppa - 3	6	87	7	75	6	89	7	81

Table 4.7 Percentage of branches that are not predicted by either heuristic.

Benchmark - Input Pair	Ball and Larus No Predict %	Dissertation No Predict %
espresso - 1	13.2	8.6
espresso - 2	13.7	18.6
espresso - 3	16.2	16.2
sc - 1	24.4	31.1
sc - 2	19.4	29.1
sc - 3	16.3	24.0
go - 1	20.5	42.7
go - 2	20.2	42.3
go - 3	19.9	42.3
m88ksim - 1	31.8	38.4
m88ksim - 2	32.2	39.2
m88ksim - 3	30.5	36.3
gcc - 1	9.5	12.8
gcc - 2	10.9	13.3
gcc - 3	7.7	11.8
compress - 1	10.5	25.4
compress - 2	5.6	13.2
compress - 3	14.4	35.1
li - 1	3.4	4.3
li - 2	2.1	3.0
li - 3	2.8	4.0
ijpeg - 1	18.1	29.6
ijpeg - 2	16.3	30.9
ijpeg - 3	24.7	38.7
perl - 1	8.1	12.6
perl - 2	3.8	6.6
perl - 3	3.9	8.9
vortex - 1	24.0	40.1
vortex - 2	27.5	40.9
vortex - 3	28.8	41.4
Pcode - 1	10.7	15.2
Pcode - 2	11.1	15.9
Pcode - 3	12.5	20.4
Lhppa - 1	4.0	4.0
Lhppa - 2	3.7	16.0
Lhppa - 3	4.6	2.8

Now the prevalence of branches that do not have any heuristics applied to them, and therefore have no predictions made for them, are investigated. This is a serious issue because the compiler is given no hints as to how these branches may behave for real program executions. The percentage of the total branches in the program that do not have a prediction applied to them are presented in Table 4.7.

The branch prediction heuristics used in this dissertation and the heuristics used by Ball and Larus are not able to predict all the branches because they depend heavily on a branch's context in the code to allow the heuristics to make their predictions. For example, one branch target must contain a subroutine call or be a loop header in order for a prediction to be made. Many branches do not contain the necessary context to allow a prediction to be made. As mentioned earlier, this same problem affects branches that are derived from `switch` statements.

For the most part, the Ball and Larus heuristics are able to predict more branches. This happens because the Ball and Larus guard and store heuristics, which are not used in this dissertation's set of heuristics, can predict a larger number of branches. There are only two benchmark-input pairs where this dissertation's heuristics are able to predict more branches than the Ball and Larus heuristics. The cases are *espresso*-input1 and *Lhppa*-input3, and they occur because of the character comparison and restricted opcode AND heuristics, which are not used in the Ball and Larus heuristics.

4.3 Loop-Trip-Count Prediction

The static prediction of loop trip counts is important if a compiler is to apply aggressive ILP optimizations. It is important to know the trip count behavior of loops so that the correct trade-offs between latency and throughput can be made. Unfortunately, the static prediction of loop trip counts is difficult. This section looks at previous work in this area, discusses the issues involved with static loop-trip-count prediction, and details some potential heuristics that can be used.

4.3.1 Previous work

Even though the static prediction of loops is an important problem, it has been largely ignored in literature. Even papers dealing with the static generation of program frequency information have ignored this issue. Two of the more recent papers on static frequency generation assume that all loops iterate the same number of times when computing the program frequencies [30], [34].

Counted loops with constant initialization, increment, and loop bound can have their trip counts determined statically by the compiler. For these known loop-trip-count loops, the compiler can trivially determine the trip counts, but these loops account for a small percentage of the total loops found in general-purpose programs.

4.3.2 Issues

The static prediction of loop trip counts is difficult because the iteration of many loops is totally dependent on the input data, and no good, static inferences can easily be made about the content of the data. This is especially true for counted loops and linked list loops. As shown in Section 5.3.2, counted loops and linked list loops account for a large percentage of the loops in most programs. This makes the static prediction of loop trip counts in general-purpose programs extremely difficult.

Even though the static prediction of loop trip counts is difficult in general, there may be some problem domains that should be able to benefit from static loop-trip-count prediction. These applications may contain loops that always iterate many times, or they may contain a large percentage of counted loops with known loop trip counts. In addition, there are certain types of loops where trip count prediction heuristics can be developed (e.g., loops involved in

argument processing, file processing, and string processing), and applications dominated by these loop types would be more predictable.

4.3.3 Potential heuristics

There are certain types of loops where the prediction of a loop trip count is possible. This section evaluates the issues involved with the generation of heuristics for three loop types: argument processing loops, file processing loops, and string processing loops. For the most part, loops associated with argument and string processing iterate a few times, and loops associated with file processing iterate many times.

Generation of heuristics for these cases is nontrivial. One problem is that there are so many ways to implement different functionality that designing a set of heuristics to capture all the desired cases is extremely difficult. One example of this is the use of special-purpose functions instead of the use of library routines. In order to read data from a file, the library-supported macro *getchar* or the library functions *read*, *fread*, *gets*, or *fgets* can be used. However, the UNIX utility *tbl* defines its own function, *gets1*, to read data from a file. This function is similar to *gets*, except that it removes '\n' from the returned string and performs other special-purpose functionality. Since a library routine is not used directly, it is impossible to capture file processing loops using this function in a heuristic without performing a search to determine whether user-defined functions are performing equivalent functionality to a library routine.

Another implementation issue occurs when a completely different method of determining a condition is used. For example, most programs, when they are determining whether a character is a space, use the library character testing function *isspace*. However, the UNIX utility *cccp* does not use this function or any user-written equivalent function. In *cccp*, a '\n' is not

considered a space character that can be skipped. To determine if a character is a space that can be skipped, it accesses a character array with 256 elements. The array contains a one in all the indices corresponding to the space characters that can be skipped, and a zero in all other locations. In this way, one load can replace several branches if a function was used to perform the same functionality. The loops controlled by this array in *cccp* account for 24% of the loops contained in the source code. Special-case situations like this must be handled, and even anticipated, if a heuristic is going to capture all the relevant cases.

Another difficulty for string processing loops deals with determining whether constant integer values are derived from constant character values in the source code, and not enumerated type or constant integer values. As mentioned in Section 4.2.2.7, the IMPACT EDG front end converts all character and enumerated type constants to integer constants. Therefore, many string processing loops must be guessed at because they are completely controlled by checks against integer constants (e.g., before going through the EDG front end, a loop may be "`while (char_variable != '\') do_something()`" in the source code, and "`while (char_variable != 10) do_something()`" after going through the EDG front end). A similar issue occurs in loops that iterate over character variables passed in as parameters. If a heuristic is going to predict these loops as string processing loops, the concept that the loops are iterating over characters must be determined.

As the previous paragraphs illustrate, the creation of heuristics that capture all the cases associated with these loop types is difficult. To handle these cases, *ad hoc* heuristics must be developed so that an implementation can be efficient in both memory and time. The goals of the heuristics should be to capture as many relevant cases as possible while minimizing extraneous

cases. The next three subsections describe some concepts that may be useful when designing heuristics to predict the loop trip count for specific loop types.

4.3.3.1 Argument processing

Argument processing loops iterate over the arguments that are present when a program is invoked. In C programs, the `argc` and `argv` variables passed into the function *main* contain the information for the arguments passed into the program. Argument processing loops are easy to determine, but programs do not contain many of these loops.

In order to determine that a loop is an argument processing loop, the function with the loop must have `argc` and `argv` variables as its incoming parameters or access global variables containing their values. If a program uses arguments, the function *main* has these variables. It is possible for another routine to actually process the arguments, but there should be a path through the call graph from *main*, and the `argc` and `argv` variables should be passed as parameters through all the call sites in the call-graph path, or global variables containing their values should be read.

If a function can access `argc` and `argv`, then all the loops in the function can be examined to see if they are performing argument processing. To determine whether a loop is performing argument processing, the branches associated with the non-error exits of the loop should meet one of the following conditions.

- One operand is `argc`, it is loop invariant, and the other operand increments by one each loop iteration.
- One operand is `argc`, it increments by one each loop iteration, and the other operand is a constant integer.

- One operand is derived from a chain of two loads, and the first load uses `argv` as the base address.

If an argument processing loop is detected, the loop should be predicted to have a short trip count.

4.3.3.2 File processing

File processing loops iterate over a file's contents. These loops read and write data to a file. Determining file processing loops can be tricky because loops may perform file I/O through user-defined functions instead of library routines. This situation is described at the beginning of Section 4.3.3.

To determine that a loop is performing file processing, the branches associated with the non-error exits of the loops should meet one of the following conditions.

- One operand is negative one (for EOF), and the other operand results from a subroutine call that reads a character (e.g., *filbuf* for the *getchar* macro or an equivalent user-defined function).

- One operand is zero, and the other operand results from a subroutine call that reads strings from a file (e.g., *read*, *gets*, or an equivalent user-defined function).

If a file processing loop is detected, the loop should be predicted to have a high trip count.

4.3.3.3 String processing

String processing loops iterate over characters. These loops are prevalent in programs that read and write ASCII text, but they can be found in most programs. There is a big payoff

if these loops can be detected because they can be an important and prevalent loop type. Unfortunately, detecting all the string processing cases and ensuring that extraneous cases are not detected can be extremely difficult. Many of these issues are discussed at the beginning of Section 4.3.3.

To determine that a loop is performing string processing, the branches associated with the non-error exits of the loop should meet one of the following conditions.

- One operand is a character constant, and the other operand is a variable that contains a character.
- One operand is zero, and the other operand results from a subroutine call that performs character testing (e.g., *isspace*, *isdigit*, or any other character testing library routine or user-defined function).
- One operand gets incremented by one each loop iteration, and it is used when accessing a character array.

If a string processing loop is detected, the loop should be predicted to have a short trip count. There are several cases where string processing loops have a high trip count. For example, a loop may iterate over all characters until the end of a comment is reached. If the input file contains a large amount of text in comments, the loop ends up having a high trip count. In these cases, the heuristic makes an incorrect prediction.

One subtype of string processing loops that should be detected is loops that iterate over white space. Almost always, these loops iterate few times. Therefore, the confidence in a prediction for a loop iterating over white space is higher than the confidence in a normal string

processing prediction. These loops can also be located all over the source code. In *cccp*, one quarter of the loops in the source code skip over white space.

4.4 Static Frequency Generation

The static generation of program frequency information is desirable because the annotation of basic block and edge frequencies is a common way for the behavior observed during profiling to be expressed to the compiler. If the results of static branch and loop-trip-count-prediction could be expressed in the same way, the same compiler heuristics may be able to work at all times, independent of whether profiling or static analysis is performed. Unfortunately, the inaccuracies and difficulties in statically predicting the branch directions and loop trip counts make the use of the same compiler heuristics in both cases unlikely to ever occur.

This section investigates the issues associated with static frequency generation. First, a summary of the previous work in this area is examined. Then a discussion of the issues involved with static frequency generation is performed.

4.4.1 Previous work

In the past, static frequency generation techniques have been looked at for several reasons. Early work was geared towards estimating the running time of a program and understanding how the program should be laid out in memory. More recent work has been geared towards providing the compiler with static frequency information that is similar to the frequency information obtained from profiling. Most of this work has used one of two different methods. One method is based on *Kirchoff's current law* (KCL), and the other method treats the program as a *discrete-parameter Markov chain*.

The KCL approach is performed on a program flow chart [36], [37]. For a program flow chart, KCL is defined to be the "sum of E's entering box = frequency of box = sum of E's leaving box," where E is an edge in the flow chart and box corresponds to a node in the flow chart. The unknowns are the edge frequencies, and KCL is used to determine a minimum set of equations and unknowns needed to solve the problem. Once estimates for the edge frequencies are made, the running time of the program can be computed. Unfortunately, the estimation of the edge frequencies can be extremely difficult to obtain [37].

The other approach to static frequency generation treats the program as a discrete-parameter Markov chain [38], [39], [40], [37], [41]. The major assumption made in this work is that the branch probabilities found in the program are independent of one another. This inaccurate yet simple model allows the use of homogeneous Markov chains. The Markov chain approach is desirable because it is easier to estimate branch probabilities than to estimate edge frequencies [39].

The first person to discuss static frequency generation for guiding the optimizations in a compiler was Wall [42]. He developed several static frequency generation schemes to see how well static estimation could match estimates based on profiling. For his work, he used some naive heuristics based on the number of loops containing an instruction, the distance of a procedure from the most distant leaf in the callgraph, and the number of direct procedure calls contained in the entire program. He did not use the KCL or discrete-Markov chain methods, and his results were poor.

Wu and Larus generate static program frequencies so that a compiler is able to identify the important code segments for optimization [30]. The first part of their work determines the branch probabilities that should be used for generating static frequencies. They use the

miss rates presented in a paper by Ball and Larus [26] for the probabilities associated with each heuristic, and they use the Dempster-Shafer theory of evidence [31] to combine the probabilities. The combined probabilities are then used in the determination of static frequencies. Problems associated with the use of the Dempster-Shafer theory of evidence in this situation are discussed in Section 4.2.1.

The second part of the Wu and Larus work is the definition of two $O(n^2)$ algorithms to perform local (intraprocedural) and global (interprocedural) static frequency generation. These algorithms are meant to be fast and always generate a solution. However, there are some serious issues with these algorithms. A discussion of these issues is presented in Section 4.4.2.

Wagner et al. generate static program frequencies for the same reason as Wu and Larus: to allow the compiler to identify the important code segments for optimization [34]. They recommend using a simple loop-based technique to generate the local frequencies, and using a Markov model of control flow for global frequencies. The simple loop-based technique assumes all branches are equally likely, all loops iterate five times, and control flow from `break`, `continue`, `goto`, and `return` statements is ignored. They used a more sophisticated branch prediction scheme using abstract syntax tree-based heuristics, but using the more sophisticated heuristics did not provide a significant performance increase over the assumption that all branches are equally likely.

The Markov model of control flow for global frequencies is similar to the discrete-parameter Markov chain method used in previous work. A major issue with this work is that they do not use probabilities when solving the Markov chain problem. This problem is discussed in more detail in Section 4.4.2.

4.4.2 Issues

There are several issues with the static frequency generation work presented by Wu and Larus, and Wagner et al. that needs to be addressed. The first has to do with the methods used by each set of authors to generate global frequencies for an entire program. Both methods leverage off the work of the previous discrete-parameter Markov chains. Wu and Larus use the same underlying equations to generate their order $O(n^2)$ algorithms, and Wagner et al. use a Markov model for their solution.

The problem is that both methods do not use probabilities when solving for global frequencies, and probabilities are a necessary component of a Markov model. When they solve for local frequencies, they use the probabilities associated with branches, and they do not run into any problems. However, when they solve for the global frequencies, they use relative call frequencies instead of probabilities. A relative call frequency is defined to be the number of times that a function, *foo*, calls another function, *foo_other*, assuming *foo* is executed once. These normalized call frequencies can have values greater than one, and this causes problems mathematically.

There is also an issue with the local frequency algorithm proposed by Wu and Larus. This algorithm does not perform well when a function contains irreducible loops. While most functions do not contain irreducible loops, some do. One function that does contain an irreducible loop is *yyparse* from the SPEC CINT92 benchmark *cc1*. A simplified example of the irreducible loop control flow found in this program is shown in Figure 4.9.

Assuming that node 1 has an execution frequency of 1, the execution frequencies are 1.4, 1.2, and 1 for nodes 2, 3, and 4, respectively. The Wu and Larus local algorithm ends up not providing any frequencies for nodes 2, 3, and 4 because of its assumptions concerning

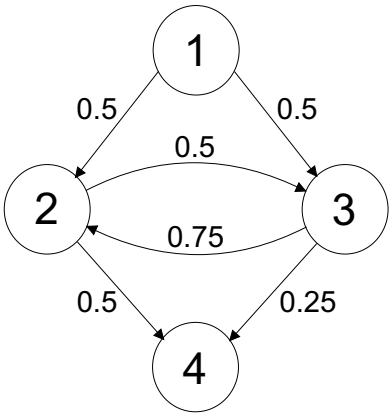

Figure 4.9 Example of the irreducible loop found in *cc1*'s function *yyparse*.

reducible loops. If a compiler is going to depend on static frequency information to help guide the compilation, not providing frequency information in certain instances is unacceptable. The Wu and Larus global algorithm does handle irreducible loops better, and ends up generating frequencies of 2, 1, and 1.25 for nodes 2, 3, and 4 if it is applied to the same code.

One final note on the entire problem of static frequency generation is that most of the methods presented previously are more complex than necessary. The lack of reliable branch prediction for a large number of branches, and the difficulty in producing loop-trip-count predictions, means that any solution for static program frequencies will not produce accurate results. Since the results will not be accurate, using a Markov model to solve this problem seems overly complex. While several authors have presented alternative techniques for solving the static frequency generation problem [42], [30], [34], most of the methods are too simplistic. The Wu and Larus algorithms try to provide decreased complexity while providing slightly less accurate results, but as shown earlier in this section, there are several assumptions made by their algorithms that can cause problems.

4.5 Conclusion

This chapter has investigated the issues associated with compilation when no profiling is performed. This dissertation proposes static analysis techniques to infer the information normally obtained by profiling. This includes the static prediction of branch direction and the static prediction of loop trip counts. The idea is for this information to be available to the compiler's heuristics so they can best determine how to apply the aggressive ILP optimizations.

The state of the art in static branch prediction has been improved by the work presented in this chapter. Improvements to several of the heuristics proposed by Ball and Larus are presented, new heuristics are proposed, and insight into the working mechanism behind these heuristics is provided. The problem of static loop-trip-count prediction is also investigated. While no working heuristics are provided, insight into what makes this problem so difficult and hints on several potential heuristics are shown. Finally, the issue of static frequency generation is investigated.

As seen in this chapter, the static prediction of branch direction and loop trip counts is difficult. When it is possible for the static analysis techniques to make predictions, the compiler has an opportunity to take advantage of it. However, static analysis techniques are not always able to make predictions. Because of this fact, aggressive ILP compilers need to have profiling performed on a program in order to obtain the best run-time performance.

CHAPTER 5

LOOP GROUPING

As discussed in Section 4.3, the static prediction of loop trip counts is difficult for many programs. This difficulty stems from most loops being controlled by the input data, and from the content of the input data not being known until the program is run. Without knowledge of the input data, the compiler is limited in the predictions that can be made about loop trip counts. Knowledge of loop trip counts for a program is important to the compiler. It allows the compiler to successfully apply optimizations like function inlining, loop unrolling, and software pipelining. It also allows the compiler to ensure that most of its available resources are spent on important sections of code.

Profiling the program aids the determination of relative loop trip counts by measuring how the program behaves for some test inputs. However, the optimization of code that is unexercised during program profiling faces the same static loop-trip-count prediction issues present in unprofiled programs. Although most of the compiler's resources should be spent on the parts of the program exercised during program profiling, a compiler cannot ignore the optimization of unexercised program portions because they may be important for an input not used during profiling. The potential problem of unexercised code for the benchmark programs used in this dissertation is examined in Section 3.2.

There is one major difference between the static prediction of unexercised code and the static prediction discussed in Chapter 4: profiling has been performed in the unexercised code case. Since the program is profiled, the compiler has some knowledge about the input data.

The extraction of this knowledge and its application to the static prediction of the unexercised code are major contributions of this dissertation. The remainder of this chapter describes loop grouping, which is the critical technology necessary for the static prediction of loop trip counts in unexercised loops. The next chapter discusses the application of the loop grouping technology to the static trip count prediction of unexercised loops.

5.1 Overview

Loops are grouped by identifying characteristics in loop control that allow a relationship between their trip counts to be made. Relationships exist between loops located in different program locations because programs often traverse over similar data multiple times. Therefore, the behavior of one loop can be used to imply the behavior of the other loops that iterate over similar data.

A good example of this phenomenon is found in the LALR parser generator program, *yacc*. In this UNIX program, six globals control over 30% of the loops found in the original source code. A description of these globals and a measurement of the number of static and dynamic loops involved is located in Table 5.1.

Each of these globals is used as an end condition for a counted loop, and each loop iterates over a statically allocated array. Two code examples of this data traversal are located in Figure 5.1. The actual number of loops contained in the source code and controlled by each global is present in the *number of loops* column of Table 5.1. The importance of the loops during program execution is measured in the *perc. of total dyn. iterations* column. Every loop iteration taken in the program during an execution is counted once, and the values shown in the table represent the percentage of total loop iterations taken in the loops controlled by the globals

Table 5.1 Evaluation of globals used to control loops in *yacc*.

Name	Description	Number of Loops	Perc. of Total Dyn. Iterations		
			Input1	Input2	Input3
ntokens	Number of terminals in the grammar	10	3.2	3.7	3.3
nnonter	Number of nonterminals in the grammar	15	2.0	0.2	3.3
nprod	Number of production rules in the grammar	6	1.9	0.2	3.7
tbitset	Number of integers needed to hold one bit for each terminal	5	2.3	6.7	2.4
nstate	Number of states defined by the grammar	10	15.2	5.0	11.0
cwp	Working set used to generate closure for each state	10	29.5	32.1	24.4

compared to the total loop iterations taken in the entire program. This same measurement is used for the results presented in Section 3.2. The three inputs applied to *yacc* include two different C parsers (inputs 1 and 3) and a simple math expression parser (input 2). The loops controlled by these six globals account for 54%, 48%, and 47% of the total loop iterations seen when applying inputs 1, 2, and 3, respectively.

The data that can be traversed multiple times goes beyond the data that is explicitly present in a program's input and output. It also applies to data used in support of a program's manipulation of data. This is illustrated with the IMPACT program, *Lhppa*. Important data that control at least twenty loops in Lhppa are shown in Table 5.2.

The table contains four columns. The *name* column contains the name of the data controlling the loops, and the *description* column describes the use of the loops. The *loop type* and *loop category* columns correspond to the loop classifications used in this chapter. Definition of the terms found in these columns are located in Sections 5.2.2 and 5.2.3.

```
for( j=0; j<=nstate; ++j ){
        if( tystate[j] != 0 && tystate[j] != best ){
                fprintf( ftemp, "%d,%d," j, tystate[j] );
                zzgoent += 1;
                }
        }
```

(a)

```
for( i=0; i<nstate; ++i ) if( greed[i] >= max ){
        max = greed[i];
        maxi = i;
        }
```

(b)

Figure 5.1 Example of globally-controlled counted loops from *yacc*: (a) from the function *go2out* in y3.c, (b) from the function *nxti* in y4.c.

The first five rows of the table represent data that are used to directly manipulate IMPACT's Lcode internal representation. The remaining rows in the table are cases of data manipulated by *Lhppa* that are used in support of *Lhppa*'s functionality, and do not reflect anything directly present in the data input into the program. The `L_Loop` data structure contains information to manipulate loops; the next two data, `List_next` and `Set->max_base`, are used during data-flow analysis; and the final three data are all used in support of dependence graph generation. It is this rich use of data that is iterated many times that allows for the successful use of the loop grouping technology.

5.1.1 Loop types to group

Loop grouping is done on counted loops and linked list loops. Several reasons make the grouping of counted loops and linked list loops interesting. Most of the loops found in the benchmarks used in this dissertation belong to the counted loop and linked list loop categories.

Table 5.2 Data that controls multiple loops in *Lhppa*.

Name	Loop Type	Loop Category	Description
L_Cb	linked list	structure field	Code unit that represents basic blocks, superblocks, and hyperblocks. All cbs in each function are linked together.
L_Oper	linked list	structure field	Operations in the program. All opers in each cb are linked together.
L_Flow	linked list	structure field	Arcs that represent control flow between cbs. All incoming flows for a cb and outgoing flows from a cb are linked together.
L_Attr	linked list	structure field	Data located on functions, cbs, and opers that is used for informational purposes. All attributes for the same structure are linked together.
L_max_dest_operand L_max_src_operand L_max_pred_operand	counted	global	Static arrays hold all the operands contained in opers. The maximum values for each of the operand arrays are defined by these globals.
L_Loop	linked list	structure field	Information on the loops. All loops in a function are linked together.
List_next	linked list	function return value	Traversal of a general purpose list structure through a function interface.
Set->max_base	counted	structure field	Traversal of the contents of IMPACT's generic set data array of size Set->max_base.
L_Dep	linked list	structure field	All dependences going into and out of a single operation are linked together.
Dep_Info	linked list	structure field	Dependence information created for each operation. As part of dependence graph generation, these structures can be linked together.
L_conflicting_operands	counted	function return value	This function returns the number of conflicting operands that need to be accounted for. An array passed by reference contains the actual data.

In addition, counted loops and linked list loops have well defined behavior. They are also the types of loops that are good candidates for iterating over data in multiple locations.

A simple example of a counted loop is shown in Figure 5.2(a). The loop is controlled with the variable `i`, and it iterates `end` times. The determination of counted loops is discussed in Section 5.2.2. A simple example of a linked list loop is shown in Figure 5.2(b). The loop is

```
for (i = 0; i < end; i++)
    foo(array[i]);
```
(a)

```
for (ptr = start_ptr; ptr; ptr = ptr-> next)
    foo1(ptr);
```
(b)

```
for (i = 0; i < end; i++)
    a = foo(array[i]);
    if (a) break;
```
(c)

```
a = 0;
for (i = 0; !a && i < end; i++)
    a = foo(array[i]);
```
(d)

```
for (i = start; i < end; i++)
    foo(array[i]);
```
(e)

```
for (ptr = start_ptr; ptr-> field; ptr = ptr-> next)
    foo1(ptr);
```
(f)

```
for (i = 0; i < end; i++)
    a = foo(array[i]);
    if (a) exit(-1);
```
(g)

```
for (i = 0; i < end; i++)
    a = foo(array[i]);
    if (a || exit_loop()) break;
```
(h)

Figure 5.2 Loop examples: (a) counted loop, (b) linked list loop, (c) counted loop with multiple exits, (d) counted loop with complex loop expression, (e) forced-weak-grouping counted loop, (f) forced-weak-grouping linked list loop, (g) counted loop with error processing exit, (h) counted loop with conjunctive and disjunctive exit logic.

controlled with the variable `ptr`, and it iterates using `ptr`'s `next` field. The determination of linked list loops is discussed in Section 5.2.3.

It is obvious that multiple copies of the counted loop found in Figure 5.2(a), assuming the variable `end` remains unchanged, will iterate the same number of times. It is not so clear how many times the loop found in Figure 5.2(c) will iterate relative to the Figure 5.2(a) loop. It will iterate the same number of times as long as `a` always has a value of zero, but it will iterate fewer times if `a` ever contains a nonzero value. There is a relationship between the two loops because they share a common counting mechanism, but the strength of correlation between the two loops is *weak* because the counting relationship only provides an upper bound on the number of loop iterations taken by the Figure 5.2(c) loop.

A loop group can contain loops that are strongly or weakly correlated to the group's counting or linked list mechanism. A loop is considered a member of the *strong* correlation set if the loop only has one branch controlling its iterations, and the branch is the check for a counted or linked list loop. A loop is considered a member of the *weak* correlation set if the loop has multiple branches controlling its iterations and one of the branches is the check for a counted or linked list loop.

A weak correlation set member is logically related to the other branches in a loop's exits (from now on, a branch in a loop's exit expression will be called an *exit branch*) through logical ANDs and ORs. If all the exit branches are contained in the same loop or `if` conditional expression, the logical relationship is explicitly stated in the expression. If a loop has multiple exits, like in Figure 5.2(c), the different expressions are related by an implicit AND relationship because both expressions must have a certain value for the loop to continue to iterate. This is illustrated in the nearly identical loops found in Figures 5.2(c) and (d).[1] The Figure 5.2(c) loop has two exits, one in the loop conditional expression and the other in an `if` conditional statement, while the Figure 5.2(d) loop has just one exit in its complex loop conditional expression.

There are cases when a loop has only one exit branch, but it still is considered a weak correlation set member. These are called *forced-weak-grouping* cases, and are illustrated in the Figure 5.2(e) and (f) loops. In the Figure 5.2(e) loop, the `start` condition for the counted loop is not constant. This means the loop will iterate a maximum of `end` times, assuming `start` is zero or greater. This condition is similar to the multiple exit case, where the exits are related by an AND, because both cases provide an upper bound on the number of iterations for the

[1] The only difference between the two loops is that the Figure 5.2(d) loop will execute one extra `i++` operation if `a` is the reason that the loops are exited.

loop. Figure 5.2(f) shows a linked list loop case where there is no check of `ptr` against NULL. Even though there is only one exit branch present, there is another implied branch in this loop. The programmer must know that `ptr->field` will be NULL before the end of the linked list is reached, so that the insertion of the check of `ptr` against NULL is irrelevant and unnecessary. Since the loop will iterate no more, and probably fewer times, than the linked list traversal found in Figure 5.2(b), the Figure 5.2(f) loop is considered a weak correlation set member of the same loop group that has the Figure 5.2(b) loop as a strong correlation set member.

Some simplifications have been made in this dissertation to make the analysis easier and increase the number of strong correlation set members that can be found. The first simplification is to ignore exits where error processing is taking place at the exit's target. This is done because errors are rare events that should not be seen, and therefore they should not affect the loop's trip count. The same method used to determine branches associated with call exiting and call error processing in the static branch prediction discussed in Section 4.2.2.2 is applied here. The `if` expression in the Figure 5.2(g) loop is an example of an error exit. Since the `if` expression has an error condition as its target outside of the loop, the loop is considered a strong correlation set member of a counted loop group based on the variable `end`.

Another simplification is to ignore exit branches that are too deeply nested in the logic controlling a loop's exit. If only an AND or OR relationship is present in a loop's exit control logic, assertions concerning the loop-trip-count behavior can easily be made. If an AND is present, the loop iterates no more than any exit branch will allow; if an OR relation is present, the loop iterates at least as many times as any exit branch will allow. It becomes less clear what should happen when both AND and OR relationships are present in the same loop control logic. For this reason, the syntax tree representation is used for the exit control logic, and the

loop control logic is not traversed past any node that is a different logic type than the root of the syntax tree [24]. For the Figure 5.2(h) loop, not all of the expressions are evaluated for loop grouping. The syntax tree for the loop control logic has an AND relation at its root because it has two different exit expressions. The only exit branch evaluated is the one corresponding to the loop expression, "`i<end`." Since the `if` expression contains a logical OR, the two branches associated with the `if` statement, which are located below the OR-node in the syntax tree, are not evaluated for grouping.

5.1.2 Related work

There is no previous work that attempts to analyze and group all the counted and linked list loops in a program. There has been a large amount of work devoted to loop analysis and optimization, but it has mostly been restricted to counted loops (`do` loops in FORTRAN programs) [43], [44]. In addition, this work, when it has evaluated and "grouped" loops, has only been performed on `do` loops that are located near one another.

Loop fusion is an example of one optimization where the counted loop bounds of adjacent loops are analyzed. If adjacent counted loops share the same loop bounds, the loops can be fused together. This allows the loop overhead to be reduced, increases instruction parallelism, and improves the register and cache locality [43], [44]. However, this optimization is only performed for counted loops and only gets applied to adjacent loops.

In this dissertation, all the counted and linked list loops are analyzed and potentially grouped. This allows all counted and linked list loops that share common loop control to be known at compile time. This analysis has several advantages. In this dissertation, Chapter 6 discusses how these groups enable trip count prediction for loops that are unexercised

during program profiling. In addition, Section 5.4 discusses other potential applications of this technology.

5.2 Loop Grouping Mechanism

Loop grouping is identifying loops that are related to one another. The actual grouping is done for loops that share the same counted loop or linked list loop mechanism, and it is done by evaluating each exit branch associated with the loops.

Counted and linked list loops all contain at least one branch that performs the check to determine whether the loop should continue to iterate or not. For a counted loop, the branch is a check of the counting variable against the `end` condition. For a linked list loop, the branch is a check of the iterating pointer against NULL. Each of these checks is also contained in an expression where one target of the check branch is the loop body (when the loop continues to iterate) and the other target exits the loop body (when the loop stops iterating). This allows loop grouping analysis to be performed individually on each branch associated with a loop exit.

The first part of this section describes the high-level loop grouping algorithm. The next two parts of this section describe the algorithms for determining counted loops and linked list loops. Finally, the problems with loop grouping and the enhancements needed to make loop grouping more successful are described.

5.2.1 High-level loop grouping

Loop grouping is done by evaluating the exits and their corresponding branches for each loop in the program. The loop groups are formed based on the results of this analysis. The

```
loop_grouping_initialization(program)
{
    For each function in program visited in levelized call-graph order {
        Generate syntax trees for all complex expressions in function
        Generate type access information for all loads in function
        Generate interprocedural information for all call sites in function
        Detect all loops in function
        For each loop in function {
            Link all the loop's exits via the syntax trees
            Mark the loop's error exits
        }
    }
}
```

Figure 5.3 Algorithm for loop grouping initialization.

loop grouping process is broken down into two parts: initialization and the actual loop group formation.

The algorithm used for initialization is shown in Figure 5.3. The purpose of initialization is to gather and organize the information needed for loop grouping. The first step of the initialization is the generation of data structures necessary to support the grouping. These structures include syntax trees, type access information, and interprocedural information.

Syntax trees are used to represent the relationship of branches in complex expressions [24], and they are built by accessing information obtained from the IMPACT Pcode front end. Enough information is present so that the Boolean relationship between branches that were originally contained in the same complex conditional expression can be derived from an `if` condition or the conditional expression of a loop.

Another important piece of information that is obtained from the Pcode front end is the access type for each load. The definition of the access information is described in Section 2.2.

This information is used to determine which loads are associated with aggregate data structures, structures or unions, and which fields in those data structures are being accessed.

The final information that is generated is the interprocedural information for each call site. This information allows all the possible values used in determining loop groups to be determined when the analysis shows that an incoming parameter is controlling the loop. The access and use of this information is described in Section 5.2.2.4. For efficiency reasons, the functions are visited in their levelized call-graph order. Levelized call-graph order is the same as the breadth-first search order with the function *main* used as the seed. In addition, other functions that do not have any incoming call-graph arcs are also used as seeds for further breadth-first ordering, until all the functions in the call graph are ordered. This means that if an outgoing parameter for a call site is dependent on an incoming parameter of the current function, the value of the incoming parameter should be available.

The other functionality performed during initialization is the detection of loops, the linking of all the syntax trees associated with each exit in the loop, and the marking of error exits. The linking of the syntax trees allows easy access to the exit data that will be traversed throughout the loop grouping process. If an exit consists of only a branch, a syntax tree for just the branch is created. The marking of error exits is important so that better loop grouping can occur. Error exits are described in Section 5.1.1.

After initialization is complete, the grouping of loops is performed. The high-level algorithm for loop grouping is shown in Figure 5.4. The results of loop grouping are kept in data structures that contain lists of loops that belong to the same counted or linked list loop grouping mechanism. The individual groups are created based on the types described in Sections 5.2.2.2 and 5.2.3.2, and on the actual variable or data type used in the counting or linked list mecha-

```
perform_loop_grouping(program)
{
    For each function in program {
        Perform reaching definition data-flow analysis
        For each loop in function {
            If (loop appears to have no exit) continue
            For each exit in loop {
                if (exit is not an error exit or all exits in loop are error exits)
                    Execute determine_related_for_exit(loop->root_type, exit->root)
            }
        }
    }
}

determine_related_for_exit (root_type, node)
{
    If (node is branch) {
        Execute look_for_counted_loop(branch)
        Execute look_for_linked_loop(branch)
    }
    Else if (node->type equals root_type) {
        Execute determine_related_for_exit(root_type, node->right_child)
        Execute determine_related_for_exit(root_type, node->left_child)
    }
}
```

Figure 5.4 High-level algorithm for loop grouping.

nism. Loops are evaluated individually and get added to the data structures when a counted or linked list loop determination is made. After all the loops are evaluated, a postpass is performed that eliminates any loop group that only contains one member. By definition, a group must contain at least two members.

Loop grouping is performed individually by evaluating the non-error exits for loops in each function. Two special cases must be accounted for. The first is when a loop appears to have no exit. This rarely occurs, but can happen when the only means of exiting a loop is through a call to *exit*. If no exit is detected, the loop is skipped. The second is when error processing is

present on all loop exit targets. When this occurs, error exits are treated just like any non-error exit is normally treated.

Exits are evaluated in the recursive function, *determine_related_for_exit*. This function takes two incoming parameters. The *root_type* is the type found in the root node of an exit syntax tree if the loop contains one exit or is an AND if multiple exits are present in the loop. The root_type is defined by the loop that is being evaluated, and it does not change during the evaluation of the loop. The other parameter, *node*, is the current node in the syntax tree that is to be evaluated. When *determine_related_for_exit* is called from *perform_loop_grouping*, it is the root node of the current exit being evaluated. When *determine_related_for_exit* is called recursively, it is a child node of the current node being evaluated.

The function, *determine_related_for_exit*, calls the counted loop and linked list loop detection functions when a branch is detected. If a non-branch node is detected in the syntax tree, a check is made to ensure that the node's type (either AND or OR) is the same as the exit's root_type. If a mismatch is found, the remainder of the syntax tree is ignored, based on the simplification discussed in Section 5.1.1. If the node types match, the children of the current node are evaluated.

5.2.2 Counted loops

5.2.2.1 Definition

Counted loops work by incrementing a *loop induction variable* by some value at each loop iteration and checking to see that some condition is true for that variable. As long as the condition remains true, the loop continues to iterate. The normal case for a counted loop is that the loop induction variable is initialized to an *init* value, which is normally a constant, and

1. for (i = *init*; i < *end*; i++) {...}
2. for (i = *end* − 1; i >= *init*; i−−) {...}

init = initial value to start counting when *i* increments (normally a constant)
end = value to stop counting when *i* increments (basis for grouping)

Figure 5.5 Definitions of variables used to group counted loops.

the loop induction variable is checked to ensure that it is less than some *end* value. Figure 5.5 shows the definitions of the values, *init* and *end*, where *i* is the loop induction variable.

The first loop in Figure 5.5 is the normal counted loop case. The second loop is the equivalent of the first loop, except that the loop decrements from *end* – 1 to *init* instead of incrementing from *init* to *end* – 1. Although most of the examples in this section only show loops associated with the first loop in the figure, their application to the second loop in the figure also applies.

Counted loops are grouped based on the definition of *end* because this value can uniquely identify how long a counting loop will iterate. The different types of definitions for *end* are described in the next section. These definitions are determined by searching the code prior to the loop and evaluating the operations that are used to define *end*'s value.

5.2.2.2 Types

The *end* condition for a counted loop can be obtained from several different sources. The following are the different *end* condition definitions that are proposed in this dissertation for loop grouping of counted loops.

Constant

This is the simplest definition possible for the *end* value. When the *end* value, *init* value, and increment are all constant, the loop is considered a *known loop bound* case. For the known

```
for (ci = 0, compptr = cinfo->comp_info; ci < cinfo->num_components;
        ci++, compptr++) {
    /* Invoke per-component upsample method. Notice we pass a POINTER
     * to color_buf[ci], so that fullsize_upsample can change it.
     */
    (*upsample->methods[ci]) (cinfo, compptr,
        input_buf[ci] + (*in_row_group_ctr * upsample->rowgroup_height[ci]),
        upsample->color_buf + ci);
}
```

Figure 5.6 Example of a structure-field-controlled counted loop from *ijpeg*'s function *sep_upsample* in jdsample.c.

loop bound case, the compiler can statically determine the trip count that will occur if the known loop bound branch totally controls the number of loop iterations. Even if other exit branches control the loop trip count, the known loop bound branch provides a bound on the number of possible loop iterations.

Global

The *end* value can be obtained from a global variable in the program. This is determined by discovering that the *end* value resulted from a load with a global address label. Examples of globally-controlled counted loops from the UNIX utility *yacc* are located in Figure 5.1.

Structure field

The *end* value can be obtained from a field in a structure. This is determined by finding that the *end* value resulted from a load that accesses a structure field. A structure access is determined by examining the type access information for the load that was generated during the loop grouping initialization. An example of a counted loop controlled by a structure field from the *ijpeg* benchmark is located in Figure 5.6.

```
/* detemine conflicting operands */
n_conflict = L_conflicting_operands(dest,conflict,5, prepass);
for ( j = 0 ; j < n_conflict; j++ ) {
        /* does the operand exist in hash table ?? */
        dep = L_find_dep_operand(conflict[j]);
        /* if the operand is present, simply remove it */
        if ( dep != NULL ) {
                L_remove_dep_operand(dep);
        }
}
for (j = 0; j < n_conflict; j++) {
        L_delete_operand(conflict[j]);
}
```

Figure 5.7 Example of function-controlled counted loops from *Lhppa*'s *L_compute_cross_iter_register_dependence* function in l_dependence.c.

Function return

The *end* value can also result from the return value of a function. This is determined by tracking down the source of the *end* value to a function return. An example of function-controlled counted loops from the *Lhppa* benchmark is located in Figure 5.7. In this example, the function call *L_conflicting_operands* returns the number of conflicts and places the conflicting information in the *conflict* array that is passed to it as one of its parameters.

While this *end* value type is useful in many instances, it can also lead to the grouping of unrelated loops. For example, loops that iterate over strings often call the function *strlen*, and they use the result returned by *strlen* to control the loop trip count. In these cases, grouping based on *strlen*'s result is not useful because the strings input into *strlen* probably vary in size. Since *strlen* is a common problem case, it is not allowed to be the basis of grouping. However, other calls that are specific to a program may exhibit similar behavior, and result in

the formation of poor loop groups. The general problems associated with loop grouping are discussed in Section 5.2.4.

Local variable

The *end* value may not have resulted from one of the previously described types. There are times that loops are totally controlled by variables declared in the same function body as the loops. The loop control variable gets computed at the start of the function, and the loops that it controls are located later in the function. An example of local-variable-controlled counted loops in the *Lhppa* benchmark is located in Figure 5.8. In the example, the variable `n_src` gets set in the first loop shown. The final two loops in this example then iterate the number of times determined by `n_src`.

The local variable *end* value type is also used when interprocedural analysis is only able to track the value of an incoming parameter to a local variable present in the calling function. A description of this application of the local variable *end* type is described in Section 5.2.2.4.

Complex

The *end* value does not always result from a single type. Sometimes, the *end* value is the algebraic relationship of two or more of the previously defined types, or the *end* value has several different possible definitions based on the control logic. In order to group loops found in these situations, a complex definition for an *end* condition is allowed.

The complex definition allows the types described previously to be linked to one another by an algebraic relation (add, subtract, multiply, and divide) or multiple potential definitions. The precedence order of the algebraic operators is maintained, and a canonical order is maintained for complex links where order does not matter: add, multiply, and multiple potential definitions.

```
int n_src, n_dst, dest_eq_src;
n_src = n_dst = dest_eq_src = 0;
...
for ( i = 0; i < L_max_src_operand; i++ ) {
        L_Operand *src;
        if ( !(src = oper->src[i]) ) continue;
        if (L_is_reg(src)) {
                ...
                buffer[n_src++] = vreg;
                ...
        }
        else if (L_is_macro(src)) {
                ...
                buffer[n_src++] = R_avail_macro[j].vreg_id;
                ...
        }
}
...
for ( j = 0; j < n_src; j++ )
        if ( buffer[j] == vreg ) {
                dest_eq_src = 1;
                break;
        }
...
for ( j = 0; j < n_src; j++ )
        if ( buffer[j] == R_avail_macro[j].vreg_id ) {
                dest_eq_src = 1;
                break;
}
```

Figure 5.8 Example of local-variable-controlled counted loop from *Lhppa*'s *R_process_cb* function in r_regalloc.c.

An example of a complex-controlled counted loop is shown in Figure 5.9. To make this example more readable, the macros located in the original source code have been expanded out. The variable `n` controls the loop iterations and has two possible definitions: the constant 1 or "`mode_size[x->mode] + 3) / 4`." This leads to two elements in the complex definition that are linked by multiple potential definitions: the constant 1 and the array `mode_size`.

```
n = ((regno) >= 16 ? 1 : (mode_size[x->mode] + 3) / 4);
while (--n > 0)
    {
        live[(regno + n) / 32]
            |= 1 << ((regno + n) % 32);
        is_needed |= (needed[(regno + n) / 32]
                & 1 << ((regno + n) % 32));
    }
```

Figure 5.9 Example of complex-controlled counted loop from *cc1*'s function *mark_used_regs* in flow.c.

Some simplifications have been made to the implementation of complex definitions used in this dissertation. First, for two loops to be grouped based on a complex definition, they must share the exact same definition. For example, there is another loop in the SPEC CINT92 benchmark *cc*1 that is controlled by a variable initialized as `n`, as in Figure 5.9, but an extra check is made to see if the result of the initialization is zero. If the result is zero, the loop control variable is set to one instead of the result from the `?` operator. This loop is not grouped with the loop shown in Figure 5.9 because it has two `constant 1` elements. The other simplification is that only the potential values, and not the logic controlling the values, are used to group when multiple potential definitions are present. This simplification allows loops that contain loop control variables that share the same multiple possible definitions, but are used under different circumstances, being placed in the same loop group even though their behavior is different. However, for the benchmarks used in this dissertation, this potential problem is not encountered.

```
determine_counted_loop (branch, loop)
{
    If (known loop bound case in branch) loop group branch and exit
    Look for counting operand in branch, with the other operand of branch called noncounting operand.
    counting operand must have one of the following characteristics
        1. counting operand has a constant, integral increment or decrement for each iteration of loop
        2. counting operand has a nonintegral increment which can be simplified to characteristic 1
    If (no counting operand is found in branch) exit
    Determine initial value of counting operand when loop is first entered, and simplify when simplifying
        situation exists between counting operand and noncounting operand
    If (neither counting operand initial value nor noncounting operand are constant) forced weak grouping
        present
    Search for the end variable using reaching definition data-flow analysis by looking for the counting
        operand initial value (decrement case) or noncounting operand defining operations
        (increment case)
    If (one defining operation is found)
        Execute determine_end_condition_variable (defining operation)
    Else {
        If (local variable case is detected) loop group branch
        Else if (complex case is determined) loop group branch
    }
}
```

Figure 5.10 Algorithm for detecting counted loop cases.

5.2.2.3 Algorithm for detection

The algorithm for detection of counted loops is shown in Figure 5.10. It looks for the *end* value types just discussed, and it is called from the *determine_related_for_exit* function shown in Figure 5.4. If a match is detected and the loop is determined to belong to a loop group type, the loop and branch are both placed into a loop group data structure for its *end* value and type; in Figure 5.4 this is termed "loop group *branch*."

As described in Section 5.2.1, loop grouping is performed by evaluating one loop at a time and placing candidate counted and linked list loops into data structures if a match is detected. So if no match is detected, nothing is done. Therefore, the fall-thru cases in the counted loop detection algorithm represent cases where the loop is not a counted loop. Only cases represented by "loop group *branch*" in the algorithm are considered counted loops.

In addition, if a loop contains only one exit branch (it can contain other exit branches as long as the other branches perform error processing on exit), the loop can be considered a strong correlation set member of a counted loop. Otherwise, if the loop has multiple exit branches, the loop can only be considered a weak correlation set member of a counted loop. If a forced-weak-group situation like that described in Section 5.1.1 is detected, the loop must be considered a member of the weak correlation set.

The detection of counted loops starts with the *determine_counted_loop* function. This function determines whether the incoming *branch* for the current *loop* can be considered a counting loop situation. For the algorithm, a branch associated with a counting loop consists of two operands. One operand is the *counting operand*, and it contains the induction variable that is incremented during each loop iteration. The other operand is the *noncounting operand*, and it contains the value that is used with the counting operand to terminate the loop's execution. If a valid counting operand is found, the appropriate *end* value and type are determined.

The first check in *determine_counted_loop* sees if the known loop bound case applies. This is the easiest case to check for because all the important variables in the counted loop are constants. If the known loop bound case does not apply, the counting operand is looked for in the branch. The counting operand should have a constant integral increment or decrement for each iteration, or it can be simplified to that case.

An example of a case where the counting operand needs to be simplified is shown in Figure 5.11. The loop located in the function *sf_write* is controlled by the macro *foreach_set*. This macro is defined to efficiently compute the address of the `pset_family` array location to access, as well as to determine when the loop iterations should stop. This leads to the nonintegral increment of `p` by `R->wsize`. However, the *end* value is also a function of `R->wsize`. In fact,

```
void sf_write(fp, A)
FILE *fp;
pset_family A;
{
        register pset p, last;
        fprintf(fp, "%d %d\n", A->count, A->sf_size);
        foreach_set(A, last, p)
                set_write(fp, p);
        (void) fflush(fp);
}
```

(a)

```
/* Most efficient way to look at all members of a set family */
#define foreach_set(R, last, p)
        for(p=R->data,last=p+R->count*R->wsize;p<last;p+=R->wsize)

/* Another way that also keeps the index of the current set member in i */
#define foreachi_set(R, i, p)
        for(p=R->data,i=0;i<R->count;p+=R->wsize,i++)
```

(b)

Figure 5.11 Example of integral increment disguised as a nonintegral increment from *espresso*: (a) from the file set.c, (b) from the file espresso.h.

the loop iteration control is independent of `R->wsize`. A less efficient, but equivalent, macro to *foreach_set* is *foreachi_set*, which is also present in part (b) of the figure. The simplification step needs to recognize that the nonintegral increment is present solely for efficiency reasons, and to determine that the the increment can be constant if the *end* value is modified to not contain the nonintegral increment variable in its definition.

If no counting operand with a constant, integral increment can be found, the loop's exit branch is not considered part of a counting loop. If a valid counting operand is found, the initial value of the counting operand is determined. Again, a simplification step is required if the counting operand's initial value is related to the *end* value, or noncounting operand of the branch. This is also illustrated in Figure 5.11. This simplification step changes the initial value

of `p`, as far as the loop iteration control is concerned, to be zero, and the value of `last` for loop iteration control to be `R->count`. The two simplifications discussed result in the loop grouping considering the *foreach_set* and *foreachi_set* macro loops to be equivalent.

At this point, if both the counting operand initial value and noncounting operand are non-constant, the forced-weak-grouping condition is detected. The forced-weak-grouping condition, as discussed in Section 5.1.1, means that the loop may terminate early. The determination of forced weak grouping ensures that any loop grouping that takes place puts the loop in the weak correlation set of the counting loop group.

The final part of the *determine_counted_loop* function searches for the defining operations for the *end* value. Reaching definition data-flow analysis is used to determine the operations that define the counting operand (for a counting loop that decrements) or noncounting operand (for a counting loop that increments). If one defining operation is found, the function *determine_end_condition_variable* is called. Otherwise, a check is made to see if the local variable case applies. If the local variable case does not apply, the complex case is investigated.

The function *determine_end_condition_variable*, shown in Figure 5.12 is used to determine if a counted loop type applies. It does this by checking for counted loop conditions, and if no condition applies, it searches for the operation defining the *dominating operation*'s operands. The first thing done by the function is to initialize the *dominating operation* to the incoming operation.

Four checks are made to see if a *dominating operation* meets a condition necessary for it to be the definition of the *end* value. The first check determines if the *dominating operation* is an incoming parameter. If it is, a check of the interprocedural data is performed. A description of interprocedural analysis is present in Section 5.2.2.4. If interprocedural analysis is present, the

```
determine_end_condition_variable (incoming operation)
{
    dominating operation = incoming operation
    Loop forever { \* stops when valid counted case or invalid condition are detected *\
        If (dominating operation is an incoming parameter) {
            If (valid interprocedural information is present) {
                Check into effect of parameter
                If (grouping possible) loop group branch
            }
            Exit
        }
        If (dominating operation is a load) {
            If (global or structure load is present) loop group branch
            Exit
        }
        If (dominating operation has two operands and neither is a constant) {
            If (complex case is determined) loop group branch
            Exit
        }
        If (dominating operation is a function return value) {
            If (function return value case is detected) loop group branch
            Exit
        }
        Find the next dominating operation using reaching definition data-flow analysis by searching
            for operations defining current dominating operation
        If (one operation is found) dominating operation = operation
        Else {
            If (local variable case is detected) loop group branch
            Else if (complex case is determined) loop group branch
            Exit
        }
    }
}
```

Figure 5.12 Algorithm for determining the end condition variable for a counted loop.

appropriate loop type is processed. The second check determines if the *dominating operation* is a load. If a load is detected, either the global- or structure-controlled counted loop situation is possible. The third check determines whether both operands are nonconstant. If they are both nonconstant, complex processing is assumed. Finally, a check is made to see if the *dominating operation* is a function return value, in which case a function-controlled counted loop case applies.

After all the checks of the *dominating operation* are made, the *determine_end_condition_variable* function attempts to find the next operations that define the operands in the current *dominating operation*. This process is similar to the final part of the *determine_counted_loop* function. Both of these function sections attempt to find the next defining operation to check, and if multiple defining operations are found, to investigate the local variable and complex cases.

5.2.2.4 Interprocedural analysis

The initialization of the interprocedural data used in loop grouping is discussed in Section 5.2.1. Interprocedural information is used for the grouping of counted loops when the *end* value is defined by an incoming parameter of a function.

The use of interprocedural information is important because many *end* variables are defined by incoming parameters. These variables may be from any of the counted loop types described earlier, like constants or values obtained from a load of a global or structure field. The *end* types that can be passed as parameters are constants, globals, structure fields, function return values, and local variables.

An example of the importance of interprocedural analysis is found in the SPEC CFP92 benchmark *ear*. In *ear*, most of the important loops in the program are controlled by incoming parameters, these functions are called from only one call site, and the value of the parameter when followed through all the call sites is a global. Interprocedural analysis identified these cases and resulted in the grouping of 17% of the loops in the program that ended up accounting for over 93% of the dynamic loop iterations when the program is executed. An example of

```
/* SOS - Calculate a bunch of second order sections.
 *      y(n) = a0*x(n) + a1*x(n-1) + a2*x(n-2) - b1*y(n-1) - b2*y(n-2)
 *
 *      Both input and output can be the same vector of numbers.
 */
sos(input, state1, state2, a0, a1, a2, b1, b2, output, n)
float input[ ], state1[ ], state2[ ], a0[ ], a1[ ], a2[ ], b1[ ], b2[ ], output[ ];
int n;
{
        register int i;
        register float tempin;

#include "ivdep.h"
        for (i=0;i<n;i++){
                tempin = input[i];
                output[i] = a0[i] * tempin + state1[i];
                state1[i] = a1[i] * tempin - b1[i] * output[i] + state2[i];
                state2[i] = a2[i] * tempin - b2[i] * output[i];
        }
}
```

(a)

```
sos(InputState, Sos1State, Sos2State, a0, a1, a2, b1, b2,
          InputState, EarLength);
```

(b)

Figure 5.13 Example of usefulness of interprocedural analysis from *ear*: (a) from the file earfilters.c, (b) from the function *EARSTEP* in earfilters.c.

this situation is shown in Figure 5.13. The variable `n` is passed into the function *sos* from the function *EARSTEP*, where it is set to the value of the global, `EarLength`.

There is a simplification made in the interprocedural analysis implemented as part of this dissertation. The interprocedural information is only used when all the call sites agree on the value that is passed to the incoming parameter. For example, if two sites call the same function with two different global values, the interprocedural analysis is ignored and no loop grouping is allowed.

Many times, the interprocedural analysis is not able to trace an outgoing parameter to a value other than the local variable, or register, used to store the value that is passed in the parameter. This allows loop grouping based on local variables to occur in more situations than those described in Section 5.2.2.2. A good example of this situation is found in Figure 5.14.

In this example, the start and stop variables for the two counted loops in the function *doavg* are passed as parameters, and the only place that *doavg* is called from is the function *eval*. The other five function calls contained in the same `switch` statement as *doavg* in *eval* have similarly controlled loops. Each of these functions is performed on a user selected group of rows and columns in the spreadsheet. The loops contained in these functions are grouped together because they share the same local variable in all the loop controlling function parameters.

5.2.3 Linked list loops

5.2.3.1 Definition

Linked list loops traverse data structures that are linked together by a *next_ptr*, and they start at a structure pointed to by an *init_ptr*. Definitions of the variables used in a linked list loop are given in Figure 5.15.

Linked lists are grouped based on the *next_ptr*. Linked list loops that use the same *next_ptr* linkage are grouped together. This grouping works well normally, but does not work out well for generic data structures that use the same *next_ptr*, but traverse over different data. *Subgroups* are used to overcome this problem by evaluating the *init_ptr*. Subgroups are discussed in Section 5.2.4.1.

```
double doavg(minr, minc, maxr, maxc)
int minr, minc, maxr, maxc;
{
        double v;
        register r,c,count;
        register struct ent *p;

        v = 0;
        count = 0;
        for (r = minr; r<=maxr; r++)
                for (c = minc; c<=maxc; c++)
                        if ((p = tbl[r][c]) && p->flags&is_valid) {
                                v += p->v;
                                count++;
                        }

        if (count == 0)
                return ((double) 0);

        return (v / (double)count);
}
```

(a)

```
/* the following code segment is from the function eval */
case REDUCE | '+':
case REDUCE | '*':
case REDUCE | 'a':
case REDUCE | 's':
case REDUCE | MAX:
case REDUCE | MIN:
{
        register r,c;
        register maxr, maxc;
        register minr, minc;
        maxr = e->e.r.right.vp -> row;
        maxc = e->e.r.right.vp -> col;
        minr = e->e.r.left.vp -> row;
        minc = e->e.r.left.vp -> col;
        if (minr>maxr) r = maxr, maxr = minr, minr = r;
        if (minc>maxc) c = maxc, maxc = minc, minc = c;
        switch (e->op) {
                case REDUCE | '+': return dosum(minr, minc, maxr, maxc);
                case REDUCE | '*': return doprod(minr, minc, maxr, maxc);
                case REDUCE | 'a': return doavg(minr, minc, maxr, maxc);
                case REDUCE | 's': return dostddev(minr, minc, maxr, maxc);
                case REDUCE | MAX: return domax(minr, minc, maxr, maxc);
                case REDUCE | MIN: return domin(minr, minc, maxr, maxc);
        }
}
```

(b)

Figure 5.14 Example of local-variable-controlled counted loops from *sc*: (a) the function *doavg* in interp.c, (b) code segment from the function *eval* in interp.c.

for (p = *init_ptr*; p != NULL; p = *next_ptr*) {...}

init_ptr = initial pointer value to start the linked list traversal
next_ptr = the next pointer in the list, normally obtained with "p = p–>next"
(basis for grouping)

Figure 5.15 Definitions of variables used to group linked list loops.

5.2.3.2 Types

There are four different types of linked list loops, each representing a different means of linking data structures together in a list. The linking can be done through a structure field, return value from a function, parameter passed by reference into a function, and array index for a statically-allocated linked list implementation.

Structure field

The *next_ptr* access through a structure field is the normal linked list mechanism that is used in programs, and occurs the majority of the time. This mechanism is used extensively in the IMPACT compiler when traversing the control blocks and operations used in IMPACT's Lcode internal representation. One example of the traversal of all the control blocks and operations of a function in the *Lhppa* benchmark from IMPACT is shown in Figure 5.16.

Function return value

It is also possible for the linked list data structure to be implemented in its own module and accessed through a function interface. This forces the *next_ptr* to be the return value from a function call. Since this is also likely to be used generically, the accounting of any casts that may be present for the return value is automatically performed when creating loop groups with

```
void Lsched_add_isl_attr (L_Func *fn)
{
    L_Oper *oper;
    L_Cb *cb;
    Sched_Info *sinfo;
    int i;
    L_Attr *new_attr;

    for (cb = fn->first_cb; cb; cb = cb->next_cb)
        for (oper = cb->first_op; oper; oper = oper->next_op)
            if (oper->ext) {
                sinfo = SCHED_INFO(oper);
                new_attr = L_new_attr ("isl", 2);
                L_set_int_attr_field (new_attr, 0 , sinfo->issue_time);
                L_set_int_attr_field (new_attr, 1, sinfo->issue_slot);
                oper->attr = L_concat_attr (oper->attr, new_attr);
            }
}
```

Figure 5.16 Example of the normal linked list loop from *Lhppa*'s l_schedule.c.

function return value linkage. This type of interface is used in the *Lhppa* benchmark, and its use is illustrated in Figure 5.17.

Function parameter

It is also possible for the *next_ptr* to be obtained by passing it as a function parameter passed by reference. Since the address is passed, the callee function can set the next value for the *ptr* being used to control a linked list traversal. This linkage is used in the *li* benchmark, and shown in Figure 5.18.

In this example, the loop in part (a) of the figure is controlled by the `list` pointer that is passed by reference to the function *xlevarg*. The function *xlevarg*, shown in part (b), passes this pointer to only one function, *xlarg*. The function *xlarg*, shown in part (c), then performs the *next_ptr* functionality in the source line `*pargs = (*pargs)->n_info.n_xlist.xl_cdr`.

```
/* Generate a set of instructions contained within each pf_node
   and temporarily store it in the use_gen field of the pf_node */
List_start(pred_flow->list_pf_node);
while ((pf_node = (PF_NODE *)List_next(pred_flow->list_pf_node)))
    {
        instr_set = Set_new();
        List_start(pf_node->pf_insts);
        while ((pf_inst = (PF_INST *)List_next(pf_node->pf_insts)))
            {
                instr_set = Set_add(instr_set, pf_inst->pf_oper->oper->id);
            }
        HashTable_insert(hash_pfnode_instr, pf_node->id, instr_set);
    }
```

Figure 5.17 Example of the function-return linked list loop from *Lhppa*'s function *D_instr_dominator_postdominator* in r_dataflow.c.

As in this example, most of the time when this linkage type is detected, more analysis can show that the function parameter loop group is equivalent to another loop group. In this example, the actual linkage ends up being a structure-field-controlled linked list traversal based on `xl_cdr`. This is one example of the extra analysis needed to determine equivalent situations, and more are described in Section 5.2.4.3

Array

The final linkage type for the *next_ptr* is used extensively in the *go* benchmark. The actual linkage is done through the access of array indices, not through pointers. The use of this type of linkage in the *go* benchmark is shown in Figure 5.19.

The two static arrays that exhibit this behavior in *go*, `mvnext` and `links`, have uses illustrated in the figure. The *next_ptr*-like functionality occurs because the contents of the current array contains the index of the array index to be evaluated next. The traversal terminates when

```
/* evaluate each expression */
while (list)
        val = xlevarg(&list);
                        (a)

/* xlevarg - get the next argument and evaluate it */
NODE *xlevarg(NODE **pargs)
{
        NODE ***oldstk,*val;

        /* create a new stack frame */
        oldstk = xlsave(&val,(NODE **)NULL);

        /* get the argument */
        val = xlarg(pargs);

        /* evaluate the argument */
        val = xleval(val);

        /* restore the previous stack frame */
        xlstack = oldstk;

        /* return the argument */
        return (val);
}
                        (b)

/* xlarg - get the next argument */
NODE *xlarg(NODE **pargs)
{
        NODE *arg;

        /* make sure the argument exists */
        if (!consp(*pargs))
                xlfail("too few arguments");

        /* get the argument value */
        arg = (*pargs)->n_info.n_xlist.xl_car;

        /* move the argument pointer ahead */
        *pargs = (*pargs)->n_info.n_xlist.xl_cdr;

        /* return the argument */
        return (arg);
}
                        (c)
```

Figure 5.18 Example of the function-parameter linked list loop from *li*: (a) from the function *xcond* in xlcont.c, (b) from the file xlsubr.c, (c) from the file xlsubr.c.

```
while(lptr != -1){
        board[mvs[lptr]] = g;
        lptr = mvnext[lptr];
        }
                        (a)

while(ptr2 != EOL){ /* fix neighbors */
        g2 = list[ptr2];
        lptr = grnbp[g2];
        ptr2 = links[ptr2];
        while(lptr != EOL){
                addlist(g2,&grnbp[list[lptr]]);
                lptr = links[lptr];
                }
        }
                        (b)
```

Figure 5.19 Example of the array linked list loop from *go*: (a) from the function *lcombine* in g22.c, (b) from the function *lsplit* in g22.c.

a special constant is encountered — –1 for `mvnext` and EOL for `links`. This is essentially a statically-allocated linked list implementation.

5.2.3.3 Algorithm for detection

The algorithm for detection of linked list loops is shown in Figure 5.20. It looks for the *next_ptr* linkages just discussed and is called from the *determine_related_for_exit* function shown in Figure 5.4. Like its counted loop algorithm counterpart, if a linkage match is detected, the loop and branch are placed into the loop group based on its *next_ptr*; in Figure 5.20 this is termed "loop group *branch*."

As described in Section 5.2.1, loop grouping is performed by evaluating one loop at a time, and placing candidate counted and linked list loops into data structures if a match is detected. So if no match is detected, nothing is done. Therefore, the exit cases with no loop grouping

```
determine_linked_list_loop (branch, loop)
{
    If (branch is not a pointer comparison) {
        If (neither operand in branch is loop invariant) exit
        If (array linked loop case is detected) loop group branch and exit
        Find oper which defines non-loop-invariant operand
        If (oper is a load and one operand of oper is constant) {
            Other operand of oper is the varying operand
            Forced weak grouping present
        }
        Else exit
    }
    Else { \* normal linked list case is detected *\
        If (one operand of branch is not NULL) {
            If (one operand of branch is loop invariant)
                Varying operand is the other operand of branch and forced weak grouping present
            Else exit
        }
        Else varying operand is non-NULL operand of branch
    }
    If (varying operand is a function return value) {
        If (function-return-value linked case detected) loop group branch
        Exit
    }
    Search for dominating operations defining the varying operand using reaching definition
        data-flow analysis
    If (multiple defining operations are found) exit
    If (dominating operation is not a load) exit
    If (dominating operation is a self-antidependent load)
        Constant operand is load source operand which is not the same as the destination operand
    Else {
        Check if a situation exists which can be simplified to a self-antidependent load case
        If (simplification situation does not exist) {
            If (function-parameter linked list case detected) loop group branch
            Exit
        }
    }
    If (constant operand is not a constant) exit
    If (dominating operation is not a structure access) exit
    \* if this point of the algo is reached, a normal linked list case has been detected*\
    Loop group branch
}
```

Figure 5.20 Algorithm for detecting linked list loop cases.

represent cases where the loop is not a linked list loop. Only cases represented by "loop group *branch*" in the algorithm are considered linked list loops.

In addition, if a loop contains only one exit branch (it can contain other exit branches as long as the other branches perform error processing on exit), the loop can be considered a strong correlation set member of a linked list loop. Otherwise, if the loop has multiple exit branches, the loop can only be considered a weak correlation set member of a linked list loop. If a forced-weak-group situation, as described in Section 5.1.1, is detected, the loop must be considered a member of the weak correlation set.

For the linked list detection algorithm, a branch associated with a linked list loop consists of two operands. One operand is the *non-loop-invariant operand* that normally contains the variable used to traverse over the linked list. The other operand is the *loop-invariant operand* that contains the value that is used with the non-loop-invariant operand to terminate the loop's execution. The *varying operand* in this algorithm corresponds to the variable used to traverse over the linked list. In the normal case, the varying operand results from a single load, and the load must contain a *constant operand*.

The first check in the algorithm determines if the branch comparison involves a pointer. The normal case involves a pointer comparison. If the branch does not involve a pointer, three possibilities exist: the array linkage case is present, the forced-weak-grouping linked list case applies, or the branch has nothing to do with a linked list.

The array linkage case applies if loops like the ones found in Figure 5.19 are found. For this to be detected, the branch must be an "==" or "!=" check against a constant, and the non-loop-invariant operand in the branch must be obtained from a self-antidependent array access. The forced-weak-grouping case, described in Section 5.1.1, applies if the non-loop-invariant

operand of the branch is obtained from a load with one of the load operands being constant. If neither of these conditions applies, this branch is not considered part of a linked list loop and the function is exited.

Otherwise, if the first check in the algorithm detected a pointer comparison in the branch (the normal case), other checks are then made to determine the linked list linkage type. When the pointer comparison is against NULL, a normal linked list case is detected. If the comparison does not involve NULL, one of the branch's operands must be loop invariant. If neither operand is loop invariant, this branch is not considered part of a linked list loop. Otherwise, the branch is considered a forced-weak-grouping case, and the loop must be placed in the weak correlation set if the linked list case is determined.

If the varying operand is the result of a function return value, the function-return linked list linkage type is checked. If the function-return linked list linkage case applies, the loop is added to the loop group data structure. After processing the function-return-value linked list case, the function is exited.

The operation defining the varying operand is then located using reaching definition data-flow analysis. If multiple defining operations are found or the defining operation is not a load, the branch is not a candidate for a linked list loop, and the function is exited.

If the defining load is self-antidependent, the normal case for a linked list has been detected. If no self-antidependent load is found, the case where the *next_ptr* is saved in an intermediate, temporary pointer variable is examined. This case is common when the entire linked list is being deleted from memory, and can be simplified to the normal self-antidependent load case. An example of this type of loop is shown in Figure 5.21. The variable `next` is used to temporarily hold the `next_ptr` that will be traversed in the linked list.

```
void L_delete_all_oper_hash_entry(L_Oper_Hash_Entry *list)
{
        L_Oper_Hash_Entry *ptr, *next;

        for (ptr=list; ptr!=NULL; ptr=next) {
                next = ptr->next_oper_hash;
                ptr->prev_oper_hash = NULL;
                ptr->next_oper_hash = NULL;
                L_free(L_alloc_oper_hash_entry, ptr);
        }
}
```

Figure 5.21 Example of simplified self-antidependent load situation from *Lhppa*'s l_code.c file.

If the simplification case is not present, the function parameter linkage type is tested for. After processing the function parameter linkage case, the function is exited.

Finally, the defining load must be from a structure. If it is not, the branch is not considered a linked list loop case. Otherwise, the branch is a linked list loop case and is processed accordingly.

5.2.4 Loop grouping issues

So far, the discussion of loop grouping has not covered any potential problem areas. It is important to understand the situations where loop grouping has problems in order for the loop grouping technology to be used effectively. The following are several issues that need to be considered when using loop grouping.

Generic Linked Lists – This problem occurs when the same linked list data structure is used for different data. In this case, loops may be grouped together because of their linked list linkage, but their average loop-trip-count behavior may not be similar if they iterate over fundamentally different data.

Correlation Between Weak Group Members – Weak group members may be strongly correlated to one another if they contain the exact same loop control in every loop exit. If the normal weak correlation sets are used exclusively, this fact cannot be exploited. The trip count for the loops would only be weakly related to the loop grouping mechanism. When the weak group members share the exact same loop control, they are considered strongly correlated to one another, and they can have all the benefits of strong correlation among themselves.

Different Loop Categories Are Equivalent – Different loop categories can be equivalent to one another, and keeping them separate lessens the usefulness of the loop groups. A description of the possible cases for equivalence is presented in Section 5.2.4.3.

Different Guarding Conditions – Average trip counts for different loop group instances can exhibit varying behavior when conditions guard the loop executions. An example of this phenomenon can occur in a compiler. Consider loops that iterate over the possible targets of a basic block. If the basic block ends in an unconditional branch or is a fall-thru, there is one target; if the basic block ends in a conditional branch, there are two targets; and if the basic block ends in an indirect branch, the basic block has an unbounded number of targets. If a loop iterates over all the basic blocks, its average loop trip count will probably be between one and two. If another loop is guarded so that it only iterates over basic blocks ending in an indirect branch, its average trip count may be large. This means two loops in the same loop group will have different average loop trip counts.

Data Transforms During Execution – Data can be transformed during program execution such that the average trip count seen at the start of execution is different than the behavior

seen at the end of execution. This can happen when the data contained in a linked list is initially read in and contains few members. Then, during program execution, the linked list grows and contains many members. If one loop only iterates over the linked list at the start of execution and another loop only iterates over the linked list at the end of the execution, these loops will have different average loop trip counts.

The first three issues are covered in following subsections. *Subgroups* handle generic linked lists, *exact match groups* allow some weak correlation members in the same loop group to be evaluated as a strong correlation subset, and *equivalent groups* allow equivalent loop groups to be combined. The remaining two issues, while seen at times in the benchmarks considered in this dissertation, do not cause major problems when predicting loop trip counts for unexercised group members. For this reason, these issues are not addressed. However, if the use of the loop grouping technology is to be extended beyond the use presented in this dissertation, these issues may need to be addressed.

5.2.4.1 Subgroups

Subgroups are used to handle generic linked lists, and they get created for every linked list loop group. Subgroups are formed based on where the value of *init_ptr*, shown in Figure 5.15, comes from. The same mechanism used to find the *end* condition for a counted loop is used to find the *init_ptr* value (this mechanism is defined by the function *determine_end_condition_variable* in Figure 5.12). Therefore, any counted loop category (e.g., global, structure field, complex, etc.) discussed earlier can also be a category for the *init_ptr*.

The idea behind this solution is that even if a generic linked list is being used, the pointer holding the initial value of the linked list should come from different locations for lists traversing

fundamentally different data. However, lists sharing the same initial value location probably traverse similar data. While not all initial value locations can be determined, just as in the search for the value of a counted loop's *end* condition, subgroups do allow some differentiation for loops iterating over fundamentally different data.

Subgroups are meant to be used when *inconsistent behavior* is detected. (Both short and long average trip count loops are observed in a linked list loop group.) Subgroups are then searched to see if any of them contain more consistent data then the loop group. If more consistent data is observed, the subgroup is used instead of the loop group for the subgroup loops when predicting loop trip counts. It would be ideal to determine ahead of time, prior to evaluating the profile data, that subgroups should be used, but this is a difficult problem that is beyond the scope of this dissertation.

5.2.4.2 Exact match groups

The *exact match group* functionality allows weak correlation set members of loop groups that share the exact same loop control to be strongly correlated with one another. The idea is that if multiple loops share the same complex loop control, their trip counts will be similar. This allows strong assertions to be made about their expected trip count behavior, and, as shown in Chapter 6, this functionality allows better loop-trip-count predictions to be made for many loops.

A good example of the usefulness of exact match groups is illustrated in Figure 5.14. In this code example from *sc*, the function *doavg*, shown in part (a) of the figure, contains two loops. The outer loop iterates over a range of rows, and the inner loop iterates over a range of columns. Five other functions — *dosum*, *doprod*, *dostddev*, *domax*, and *domin* — contain the

same loops. All the row loops in these six functions end up being weak correlation set members of the loop group controlled by `maxr`, and the column loops are handled similarly, except that they are controlled by `maxc`. They are weak correlation set members because they have a non-constant initialization; therefore, each loop is considered a forced-weak-grouping counted loop. However, the initialization for each of the row loops (`minr`) and column loops (`minc`) is the same. Since the initialization is identical for the forced-weak-grouping counted loops and the loops have only one branch controlling the loop iterations, an exact match situation is detected. This means the loops in the six functions can be considered strongly correlated to one another, and their loop-trip-count behavior can be considered the same.

The exact match group functionality is applied to weak correlation set members of loop groups and subgroups. The implementation of exact match groups used in this dissertation is strict, and exact match groups are only formed in specific cases. For an exact match relationship to be determined between two loops, the loops must share the exact same syntax tree for their exits, as defined in Section 5.2.1, and each branch in the same location of the syntax tree for the two loops must be a member of the same loop group. In addition, if the branch is part of a forced-weak-grouping case, additional checks must be passed.

For a forced-weak-grouping counted loop, the initialization for the variable that is counted must be identical for both loops. The initialization is determined, as in the subgroup case, by using the same mechanism used to find the *end* condition for a counted loop (defined by the function *determine_end_condition_variable* in Figure 5.12), but applying it to the variable used to initialize the counting variable in the counted loop.

For a forced-weak-grouping linked list loop, one of two checks must be passed. If the forced-weak-grouping linked list branch is performing a pointer comparison, one operand from each

branch must be loop invariant (the other operand was used to group the loop) and the two loop-invariant operands must be derived from the same source (determined by using the same mechanism used to find the *end* condition for a counted loop). If the forced-weak-grouping linked list branch is not performing a pointer comparison, one branch operand from each load must come from a preceding load, the loads must share the same load type, as described in Section 5.2.1, and the other branch operands must be derived from the same source (again, using the same mechanism used to find the *end* condition for a counted loop).

5.2.4.3 Equivalent groups

Equivalent groups are formed when two loop groups are determined to iterate similarly. It is expected that the members of the two loop groups that are combined into an equivalent group are related to one another as if their loop group was formed as one larger group from the start. It is desirable to combine as many loop groups as possible because the power of loop grouping comes from having as many related loops as possible located in the same loop group.

Determination of equivalent groups is done after all loops in the program have been evaluated and the counted loop and linked list loop groups have been formed. Equivalent groups can be formed in four different situations. The following paragraphs describe each of the situations, and how the equivalences are determined.

One equivalent group situation involves a local variable controlled counted loop. A local-variable-controlled counted loop is formed when a control variable is incremented at the start of a function, and then loops controlled by it are located later in the function. An equivalence is determined by examining the loop that is used to increment the local variable. If the increment is done in a loop, the increment is done on every loop iteration, and the loop containing the

increment is a strong correlation member of a loop group, then the loops contained in the local-variable-controlled counted loop group can be combined with the loops contained in the loop group of the loop that increments the local variable.

Another equivalent group situation also involves a local-variable-controlled counted loop, and is similar to the previous case. The difference is that instead of evaluating where the local variable is defined, the use of the local variable is evaluated. If the local variable is the return value of the function containing the local variable and the function return value is used to form a function-controlled counted loop group, the loops in the two groups can be combined to form a new group.

A third equivalent group situation involves doubly linked lists. Doubly linked lists traverse the same data, but iterate over it in two directions — forwards and backwards. Two different structure fields are used in this case. A common way of differentiating the direction pointers is by using the words *next* and *prev* in the names of the structure fields [45]. Equivalent groups are formed when two structure-field-controlled linked lists are part of the same data structure and the only difference between the field names used for their linked list linkage are the strings *next* and *prev*.

The final equivalent group situation involves function-parameter-controlled linked lists. Function-parameter linked lists are prevalent in *li*, and an example of this situation is found in Figure 5.18. As described earlier, the loop in part (a) of the figure is controlled by the `list` pointer that is passed by reference to the function *xlevarg*. Eventually, the pointer is modified in *xlarg*, shown in part (c), in the source line, `*pargs = (*pargs)->ninfo.n_xlist.xl_cdr`.

Equivalences are determined by evaluating the function that is receiving the parameter. In this example, the function *xlevarg* is evaluated. The equivalent group processing is performed

by searching the called function and determining if a more basic loop group relationship can be determined. Evaluation of the function *xlevarg* shows that a loop group consisting of the function *xlarg* and the passed parameter can be used instead. This is a more basic loop group relationship because it is detected in the called function. Further evaluation of *xlarg* shows that the function-parameter linked list group is actually equivalent to the structure-field-controlled linked list group containing `xl_cdr`.

This analysis could have been done when the function-parameter linked list situation was first discovered, but was delayed to this step for efficiency. Waiting means the analysis can be done once for each function-parameter linked list group. The alternative would be to redo the analysis for each individual loop that ended up being contained in the function-parameter linked list group.

5.3 Results

The effectiveness of loop grouping is now considered. This analysis will answer the following questions.

- How well is loop grouping applied?
- What do the groups look like?
- What is the quality of the groups that are formed?
- How well are the issues with loop grouping being addressed?

The benchmarks and inputs used in this analysis are identical to the ones described in Section 3.1. Each of the benchmark's three inputs is analyzed using the *dynamic iterations*

(each iteration taken during a program execution gets counted once), which are introduced in Section 3.2. Dynamic iterations are a useful measure because they stay constant regardless of any optimizations that may be applied, and they show what is happening for loops that are important.

5.3.1 Potential for loop grouping

Before evaluating the effectiveness of loop grouping, the break down of the potential for *strong* and *weak* correlation set loop group members is evaluated. As defined in Section 5.1.1, a loop can become a strong correlation set member if the loop contains only one branch controlling its loop iterations, ignoring loop exit branches that are present for only error checking. If a loop has multiple branches controlling its loop iterations, the loop can only become a member of a weak correlation set. Strong correlation set member loops are useful because they allow exact assertions to be made about the loop's behavior, while weak correlation set member loops only allow bounds to be placed on the loop's behavior.

This experiment measures the fraction of dynamic loop iterations taken in loops that can be considered candidates for strong or weak correlation set membership. The results are presented in Figure 5.22. The first bar for each benchmark represents the behavior seen when input one is applied, the second bar represents the behavior when input two is applied, and the third bar represents the behavior when input three is applied. The actual inputs that are used are described in Section 3.1.

As seen in the figure, many loops for the benchmarks are controlled by only one branch and are therefore candidates for strong correlation set membership. The average behavior for the benchmarks and inputs used in this experiment shows that 54% of the dynamic loop iterations

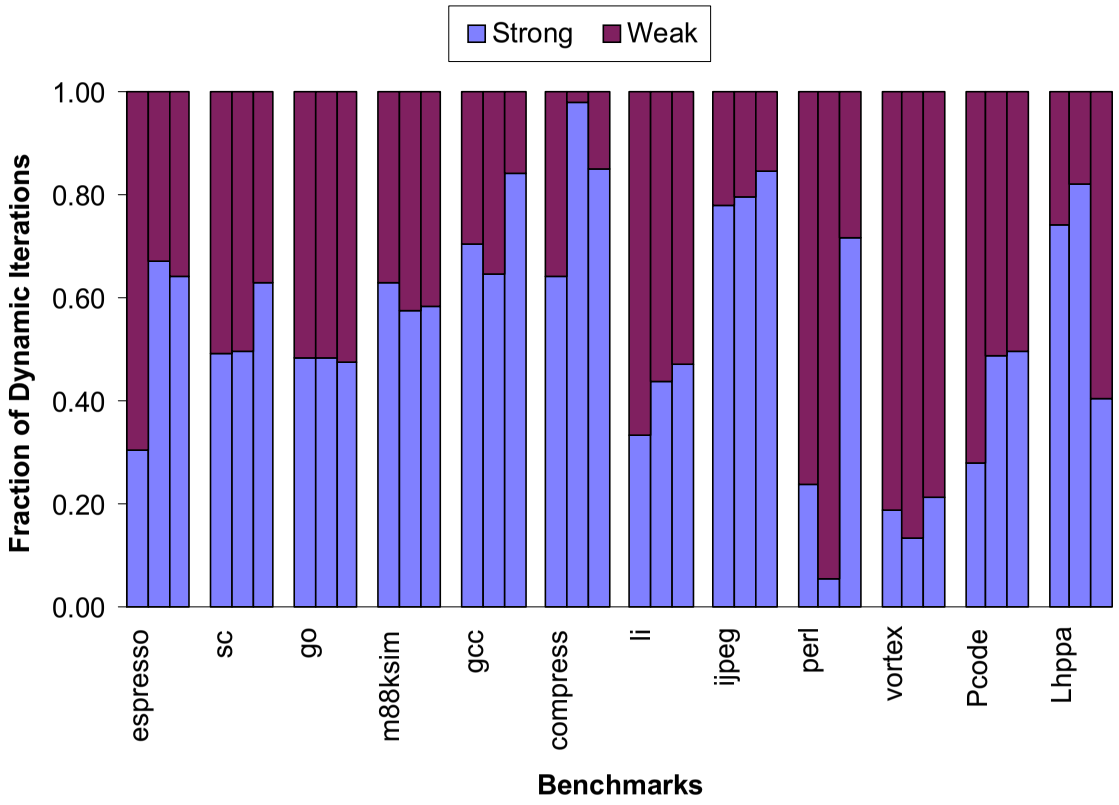

Figure 5.22 Breakdown of strong and weak candidate loops.

are spent in loops controlled by only one branch. Of the benchmark-input pairs evaluated, eight have a strong correlation set candidate percentage below 40, while 16 have a strong correlation set candidate percentage above 60. In addition, only 4 of the 12 benchmarks — *go*, *li*, *vortex*, and *Pcode* — had no input with a strong candidate percentage over 50.

5.3.2 Applicability of loop grouping

Now, the applicability, or coverage, of loop grouping is investigated. In order for loop grouping to be successful, it must be able to group a large fraction of the important loops in the benchmarks. If it is unable to do this, loop grouping is of no use.

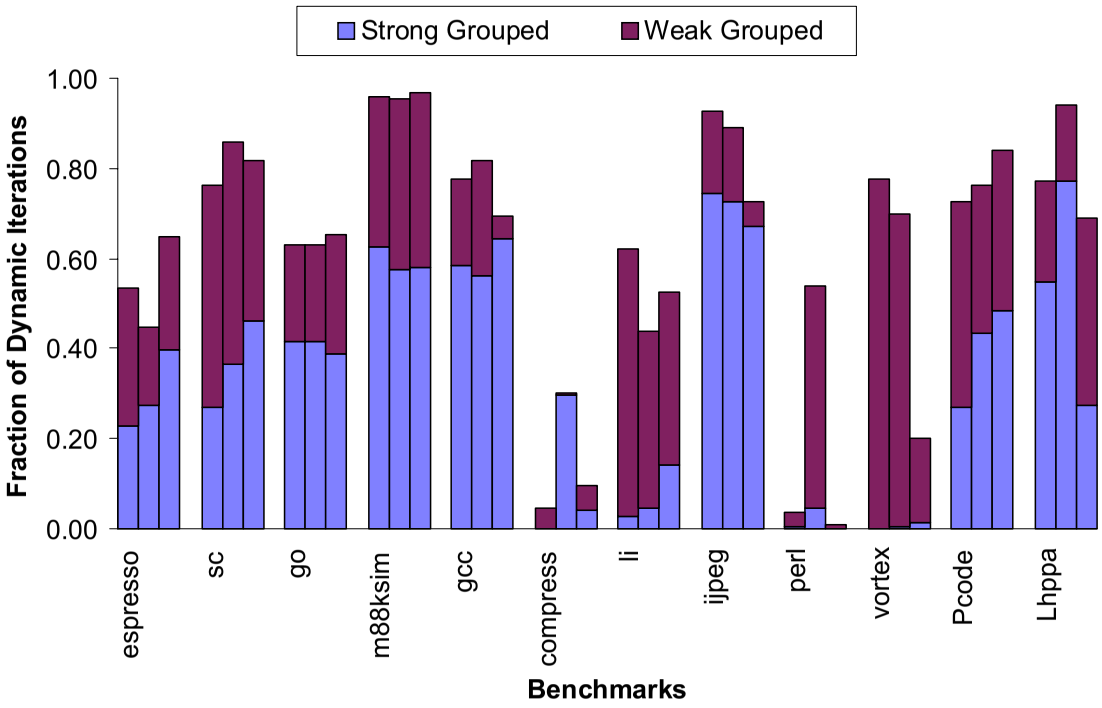

Figure 5.23 Applicability of loop grouping.

This experiment measures the fraction of dynamic loop iterations that are found in loops that are groupable. (The loops are counted and linked list loops that met the rules presented earlier in this chapter.) The results are presented in Figure 5.23, and again, each of the three bars represents the three inputs that are used throughout this dissertation (with the first bar corresponding to the first input).

As seen in the figure, for most benchmark-input pairs, a large fraction of the dynamically important loops are grouped. Twenty-five of the 36 benchmark-input pairs have over 60% of the dynamically important loops grouped, while only six benchmark-input pairs have less than 40% of the dynamically important loops grouped. These six problem benchmark-input pairs are confined to three benchmarks: *compress*, *perl*, and *vortex*. A discussion of what is happening in each of these benchmarks follows.

The benchmark *compress* has problems because the benchmark is so small. This benchmark has well-defined functionality, compression and decompression of data, and is small, containing only 24 loops. This means the loops contained in *compress* are special purpose, and there is not an opportunity for different loops to traverse with similar loop control. The only loop grouping done in *compress* occurs in counted loops with a constant end condition, which can be determined without the loop grouping technology. Loop grouping is meant to work well on large programs with rich functionality, and *compress*, being small with limited, well-defined functionality is an example of a program that does not benefit much from loop grouping.

The benchmark *perl*'s first and third input have less than 3% of their dynamically important loops grouped. These inputs have problems because of two loops. The first input's dynamic iterations are dominated by one loop that accounts for a little over 70% of the dynamic iterations. This loop is a counted loop with an end condition that can take on many different values. The definition of the variable, `offset`, which controls how long this counted loop iterates, is shown in Figure 5.24. The variable `offset` is set in six different locations, and can be algebraically related to either the global `arybase` or structure field `ary->ary_fill`. The value of `offset` has such a complex definition that the loop it controls ends up being unique, and similar loop control is not located elsewhere in the program.

The third input for *perl* has another loop that accounts for almost 60% of the dynamic iterations. This loop is an unstructured `goto` loop that cannot be grouped because it is not a counted or linked list loop. Since it is a `goto` controlled loop, there is probably not another loop that shares its same loop control, so grouping is probably not possible even if loop grouping is extended to handle loops beyond counted or linked list loops. Based on the behavior of the

```
if (++sp < max) {
        offset = (int)str_gnum(st[sp]);
        if (offset < 0)
                offset += ary->ary_fill + 1;
        else
                offset -= arybase;
        if (++sp < max) {
                length = (int)str_gnum(st[sp++]);
                if (length < 0)
                        length = 0;
        }
        else
                length = ary->ary_max + 1;          /* close enough to infinity */
}
else {
        offset = 0;
        length = ary->ary_max + 1;
}
if (offset < 0) {
        length += offset;
        offset = 0;
        if (length < 0)
                length = 0;
}
if (offset > ary->ary_fill + 1)
        offset = ary->ary_fill + 1;
```

Figure 5.24 Example of complex-end-condition problem from *perl*'s *do_splice* function in dolist.c.

loops seen for the inputs applied to *perl*, it appears that *perl* contains important loops that are unique and that do not share loop control with other loops.

The final benchmark that exhibits poor loop grouping applicability is *vortex*. Its third input is only able to group 20% of the dynamically important loops. The main difference between the inputs in *vortex* is the amount of transaction processing that is performed. Each input performs initialization, and the third input does by far the least amount of transaction processing, which means a large fraction of its execution time is spent in initialization. The

loops in initialization are not loop grouped well, so the third input has a large percentage of the dynamically important loop iterations not grouped. Two of the main reasons for the loop grouping problems during initialization are counted loop end conditions being too complex, as seen in *perl*, and loops not being the counted or linked list loop types.

With the exception of these three benchmarks, loop grouping is able to be applied to a large proportion of the dynamically important loops. The average percentage of dynamic loop iterations contained in groupable loops considering all the benchmark-input pairs is 63. If the three problem benchmarks are ignored, the percentage goes up to 74. These values are high enough that loop grouping can be a relevant technology when trying to analyze the loops in programs.

5.3.3 Loop group characteristics

The characteristics of the loop groups that are formed are now analyzed. The first experiment evaluates which counted or linked list loop categories get applied for the different benchmarks. The percentage of dynamic iterations spent in each of the categories for each benchmark-input pair is shown in Table 5.3. The categories correspond to the loop category definitions presented in Sections 5.2.2.2 and 5.2.3.2. The sum of each row in the table corresponds to the percentage of dynamic iterations that are contained in loops that are grouped, which matches the height of the corresponding bar in Figure 5.23. As seen in the table, no particular loop group category dominates all the benchmarks.

Table 5.4 shows the static characteristics of the loop groups that are formed. The information includes the number of distinct loop groups that are formed, the percentage of loops found in the source code that are grouped, and the average size of the loop groups. In addition, the

Table 5.3 Percentage of dynamic iterations spent in each of the loop grouping categories.

Benchmark - Input Pair	Counted Loops						Linked List Loops			
	Const	Global	Struct Field	Local Var	Func Ret	Complex	Struct Field	Func Ret	Func Param	Array
espresso - 1	0	0	53	0	0	0	0	0	0	0
espresso - 2	0	0	44	0	0	0	0	0	0	0
espresso - 3	0	0	63	0	0	0	2	0	0	0
sc - 1	0	51	0	8	0	17	0	0	0	0
sc - 2	0	51	0	24	0	11	0	0	0	0
sc - 3	0	60	0	8	0	13	0	0	0	0
go - 1	3	22	0	1	0	0	0	0	0	37
go - 2	3	22	0	2	0	0	0	0	0	37
go - 3	2	23	1	1	0	0	0	0	0	37
m88ksim - 1	83	0	0	0	0	12	0	0	0	0
m88ksim - 2	94	0	0	0	0	1	0	0	0	0
m88ksim - 3	49	0	0	0	0	34	14	0	0	0
gcc - 1	28	17	1	4	0	2	23	1	1	1
gcc - 2	22	29	1	4	0	2	21	1	1	1
gcc - 3	35	15	0	3	0	5	8	0	1	2
compress - 1	5	0	0	0	0	0	0	0	0	0
compress - 2	30	0	0	0	0	0	0	0	0	0
compress - 3	9	0	0	0	0	0	0	0	0	0
li - 1	0	0	2	0	0	0	61	0	0	0
li - 2	0	0	4	0	0	0	40	0	0	0
li - 3	0	0	13	0	0	0	40	0	0	0
ijpeg - 1	57	0	36	0	0	0	0	0	0	0
ijpeg - 2	60	0	29	0	0	0	0	0	0	0
ijpeg - 3	27	4	42	0	0	0	0	0	0	0
perl - 1	0	0	0	0	0	3	0	0	0	0
perl - 2	0	0	6	0	0	47	0	0	0	0
perl - 3	0	0	0	0	0	0	1	0	0	0
vortex - 1	0	77	1	0	0	0	0	0	0	0
vortex - 2	0	68	1	0	0	0	0	0	0	0
vortex - 3	1	14	5	0	0	0	0	0	0	0
Pcode - 1	0	0	19	0	0	0	51	2	0	0
Pcode - 2	0	0	2	0	0	0	73	1	0	0
Pcode - 3	32	0	25	0	0	0	27	0	0	0
Lhppa - 1	7	27	7	0	1	0	35	0	0	0
Lhppa - 2	39	12	29	0	2	0	10	2	0	0
Lhppa - 3	4	13	4	0	0	0	48	0	0	0

percentage of the grouped loops found in the source code from groups of different sizes is also shown in the table.

As was the case dynamically in Section 5.3.2, *compress*, *perl*, and *vortex* have the worst loop grouping coverage for the loops in the source code. The benchmarks *sc* and *m88ksim* have significantly better loop grouping coverage percentage for the dynamically important loops than

Table 5.4 Static characteristics of loop groups that are formed.

Benchmark	Number of Loop Groups	Percent of Total Loops Grouped	Average Size	Percent of Loops for Each Size			
				2 to 5	6 to 10	11 to 20	21+
espresso	26	63	16.9	5	4	25	66
sc	20	41	4.1	31	69	0	0
go	35	91	17.9	9	5	2	84
m88ksim	29	54	3.6	55	12	11	22
gcc	125	77	15.3	10	5	10	75
compress	5	40	2.0	100	0	0	0
li	6	49	10.0	13	9	0	78
ijpeg	57	60	5.3	31	21	26	22
perl	36	28	4.1	54	25	21	0
vortex	19	29	3.3	74	0	26	0
Pcode	116	83	12.5	11	8	16	65
Lhppa	169	86	15.2	10	8	7	75

they do for the static loops found in the source code. Most of the benchmarks also group a significant number of loops in large loop groups. Six benchmarks have over 50% of their grouped loops in groups with a size larger than 20. In addition, only *sc*, *compress*, *perl*, and *vortex* do not have at least 20% of their grouped loops in groups with a size larger than 20.

Figure 5.25 shows the percentage of dynamically important loop groups that are contained in groups of different sizes. The height of each bar corresponds to the fraction of dynamic loop iterations that are present in loops that have been grouped. The dynamic results are similar to the static results presented in Table 5.4. The exceptions are *m88ksim*, *Pcode*, and *Lhppa*, which have a larger percentage of dynamically important loops spent in loop groups with few members. The significantly higher percentage in *m88ksim* comes from the time spent in known loop-trip-count loops. In *m88ksim*, there are a small number of known loop bound, high trip count loops sharing the same constant loop trip counts.

In this dissertation, loop grouping is used to determine the trip count of loops that are unexercised by an input. This is done by evaluating all the loops contained in a loop group. If at least one loop is exercised by an input, inferences for the trip counts of loops that are

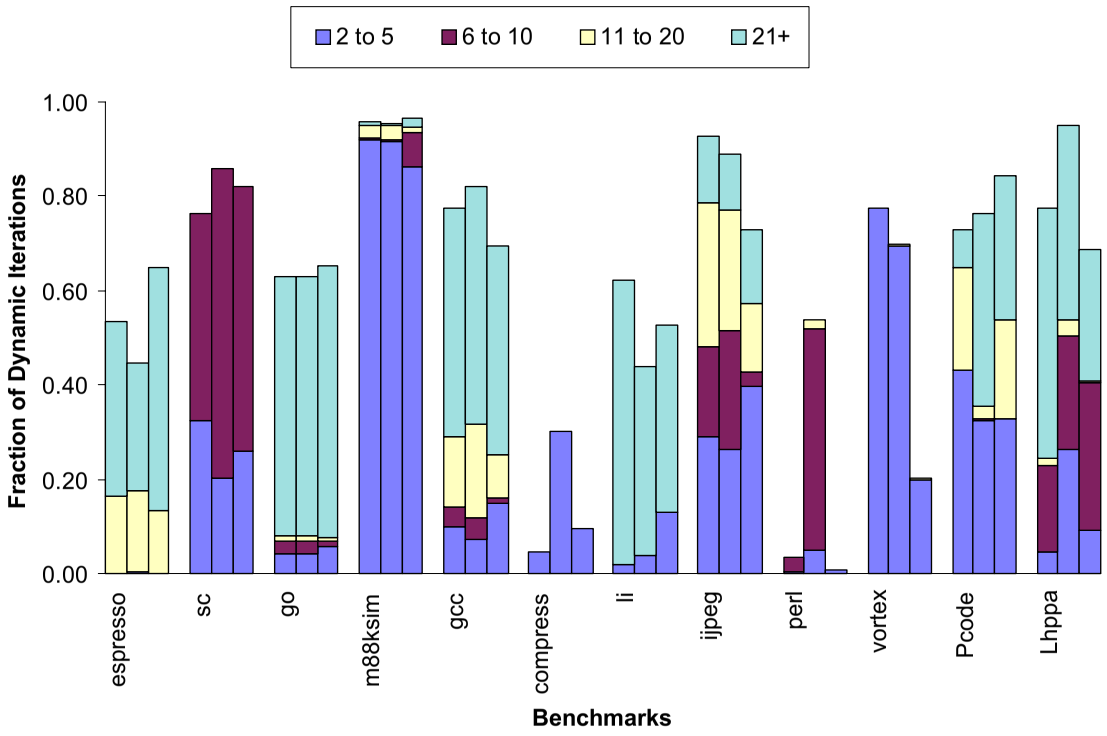

Figure 5.25 Size of loop groups in dynamically important loops.

unexercised by the same input is possible. For this to work, the trip counts observed in a loop group should be consistent. If the loop group trip counts are inconsistent (meaning that the trip counts of members of the same loop group can be short or long), then they are not useful for prediction purposes.

The next experiment evaluates how many of the dynamically important grouped loops are contained in inconsistent loop groups. An inconsistent loop group is defined to be a loop group that has two strong correlation group members, one of that has an average trip count under 5 and another that has an average trip count over 20 for at least one of the inputs used for each of the benchmarks. The results are shown in Figure 5.26. One bar is present for each input that is used for the benchmarks, the height of the bar is limited to the number of dynamic iterations present in loops that are groupable, and each bar is broken down into three categories. *Known*

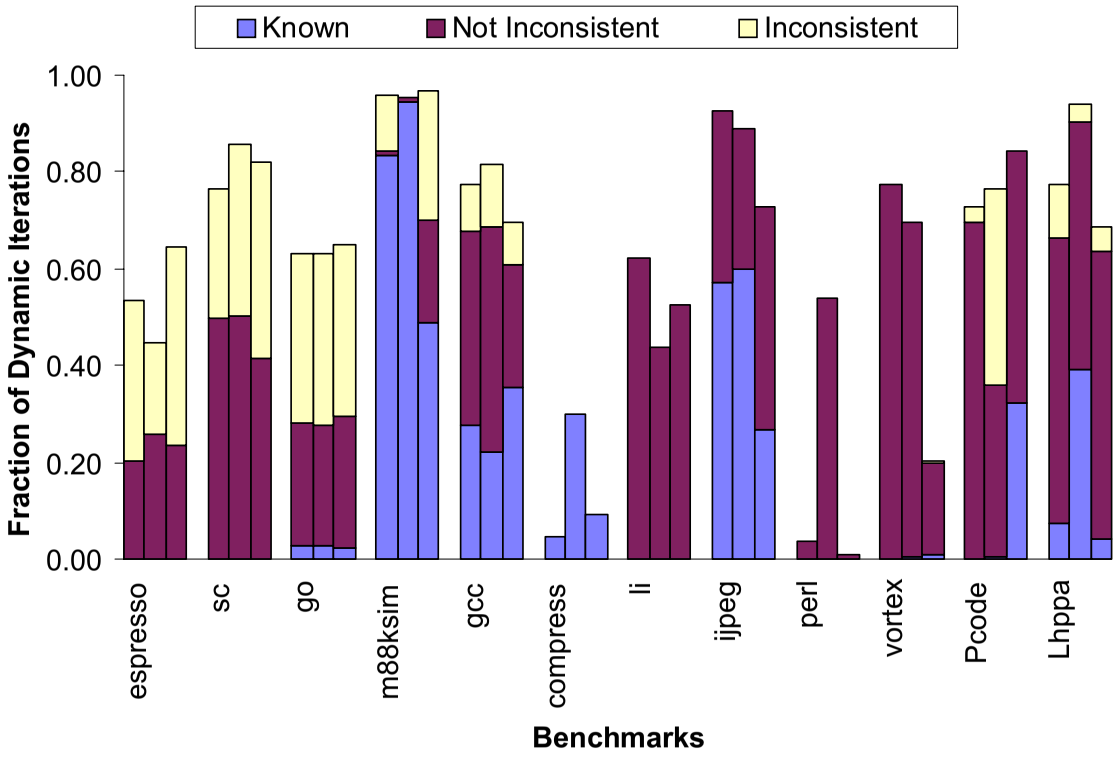

Figure 5.26 Characteristics of loops that are grouped.

means the loops are constant-controlled counted loops, *inconsistent* means that the loop group meets the definition of inconsistent presented in this paragraph, and *not inconsistent* means that the loops are not contained in loop groups that are determined to be inconsistent for at least one of the inputs.

As seen in the figure, several benchmarks have a significant amount of loop iterations present in loops that exhibit inconsistent behavior. This would not bode well for loop grouping if there were not an important trend underlying these results. There is something special happening in a majority of the inconsistent cases. In fact, for all of the inconsistent loop group cases in *espresso*, *sc*, and *m88ksim*, the grouping of loops that appear to be inconsistent may not be bad.

```
pcube *cube3list(A, B, C)
pcover A, B, C;
{
        register pcube last, p, *plist, *list;

        plist = ALLOC(pcube, A->count + B->count + C->count);
        list = plist;
        *plist++;
        last = p + A->count * A->wsize;
        for (p = A->data; p < last; p += A->wsize)
                *plist++ = p;
        last = p + B->count * B->wsize;
        for (p = B->data; p < last; p += B->wsize)
                *plist++ = p;
        last = p + C->count * C->wsize;
        for (p = C->data; p < last; p += C->wsize)
                *plist++ = p;
        *plist++ = NULL;
        list[1] = (pcube) plist;
        return list;
}
```

input 1	input 2	input 3
117	220	8
18	168	21
4	36	4

Figure 5.27 Example of inconsistent behavior found in *espresso*.

The reason that the grouping is not bad can be illustrated with the code example from *espresso* shown in Figure 5.27. The function, *cube3list*, takes in three `pcover` sets, places the contents of the sets into `list`, and the outputs `list` to the caller. To make the source code more readable, the function has been macro expanded.

This function contains three loops that are almost identical, with the only difference among them being the sets that they traverse. The loops are grouped together in a structure-field-controlled counted loop — `pcover` being the type of the structure and `count` being the field that is used for loop control. The average loop trip counts for these loops for each input are also shown in Figure 5.27.

Given the loop trip counts shown, it is obvious that there are inconsistencies in the average loop-trip-count behavior. For the first input, the loop iterating over `A` has a high trip count and the loop iterating over `C` has a low trip count. For the second input, all the loops have a

high trip count, but the loop iterating over `A` again has the highest trip count. The third input acts differently because the iteration over `B` has the highest trip count, and average trip count of `A` is small.

Given the average trip counts seen for these loops and inputs, it is not clear what the compiler should do in terms of optimization. If the compiler used input 1 for profiling, should it optimize the loop iterating over `A` as if it was a high trip count loop, and optimize the loop iterating over `C` differently because it appears to be a low trip count loop? It appears, given the data that are presented, that the loops should not be optimized differently. The order in which the loops are presented to this function may be dependent on the order in which the data were presented to the benchmark. What appeared to be a high trip count loop for one input may be a low trip count loop if the same input was used but the data were presented in a different order.

This highlights another potential use of loop grouping — the overriding of profile data during compilation for loop trip counts in the presence of inconsistencies in loop trip counts. If applied judiciously, this can lead to better performance for real inputs. This work does need to account for the issues discussed in Section 5.2.4 because it exposes cases that exhibit the same inconsistent behavior for a loop group, but that show no inconsistent behavior when the input is changed.

The issues brought up in Section 5.2.4 do account for some of the inconsistencies seen in the other benchmarks shown in Figure 5.26. The benchmark *go*'s inconsistent behavior occurs for linked list loops controlled by the array, `links`. This array is used for several different purposes, and therefore exhibits inconsistent average loop-trip-count behavior. The subgroups functionality helps out in this case. The other benchmarks that have inconsistent behavior are

affected by the issues discussed in Section 5.2.4, as well as the issue seen in *espresso*'s loops found in the *cube3list* function.

5.3.4 Effectiveness of loop grouping enhancements

5.3.4.1 Subgroups

Subgroups are designed to allow generic data structures that iterate over fundamentally different data to be dealt with effectively. Subgroups are described in Section 5.2.4.1. The effectiveness of subgroups is measured by seeing how well they can differentiate the inconsistent cases, when the same loop group contains at least one loop with a low trip count and another loop with a high trip count.

The results of this experiment are presented in Figure 5.28. The height of each bar corresponds to the fraction of dynamic iterations for the *inconsistent* case presented in Figure 5.26. Three cases are measured: *no subgroup* means no subgroups are created; *subgroup - inconsistent* means subgroups are created, but the resulting subgroup is also inconsistent; and *subgroup - consistent* means subgroups that are consistent are created.

As discussed in Section 5.3.3 and illustrated with Figure 5.27, not all the inconsistent cases are present because of generic data structures. This is true for all the inconsistent cases in *espresso*, *sc*, and *m88ksim*. In addition, the large percentage of the *subgroup - inconsistent* case for *Pcode*'s input 2 is occurring for the same reason as *espresso*. The loop group causing the problem is `node.sibling`. These same loops caused the inconsistent behavior seen in the data presented in Figure 3.1. The data contained in these inputs to *Pcode* are so fundamentally different that the formation of a more consistent group is not possible.

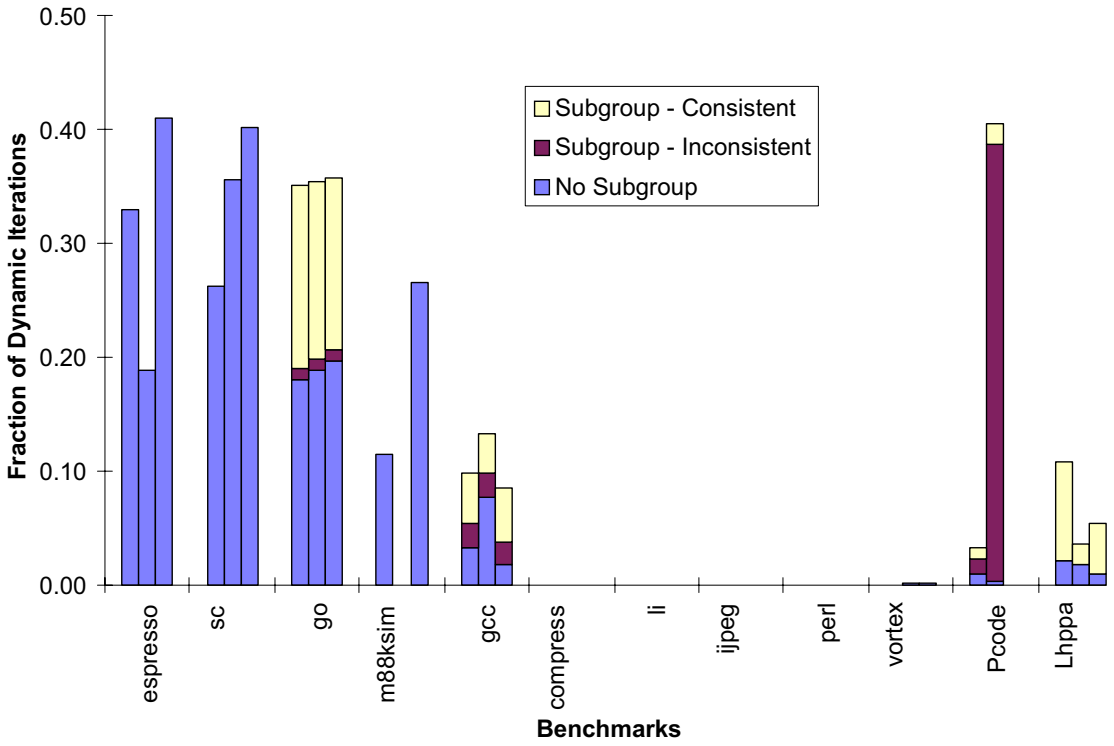

Figure 5.28 Measurement of the effectiveness of subgroups.

All the inconsistent cases in *go* result from the use of a generic data structure. One array, `links`, is used to hold many different types of data. Examples of the different data include the enemy neighbor list, list of empty neighbor squares, and connection record list. A little less than half of the loops in this inconsistent group can have their *init_ptr* resolved. Most of these subgroups are valid because they contain loops with consistent behavior.

The benchmark *gcc* is helped by the creation of some good subgroups. One of the more important subgroups is formed with the local variable `insns` from the function *rest_of_compilation*. This function is called after an input function is parsed, performs optimizations on the input function, and then outputs assembly code for the input function. The variable `insns` holds the list of all the operations contained in the function, and it ends up controlling 28 loops that use `insns` as its linked list *init_ptr*. All these loops share high average trip counts.

Lhppa is also helped by subgroups. The main benefit occurs in the linked list loop group for `L_Oper.next_op`. The inconsistencies occur when superblocks are formed. For loops that iterate over all the operations in a superblock, the average loop trip count is large. However, there are loops that only iterate over a subset of operations, and end up having a small average loop trip count. The subgroup that yields the consistent behavior has `L_Cb.first_op` as the *init_ptr*. `L_Cb.first_op` points to the first operation in a superblock, and it isolates the cases that iterate over all the operations in a superblock.

5.3.4.2 Exact match groups

Exact match groups allow weak correlation loop group members that share the same loop iteration control to be grouped together. The advantage of grouping them together is that they can be considered strongly correlated to one another, and all the benefits of strong correlation can be applied to them. The exact match group functionality is described in Section 5.2.4.2.

Exact match group success is measured by seeing how many weak correlation loop group members can be linked together with an exact match. Figure 5.29 shows the applicability of the exact match group functionality. The height of each bar corresponds to the fraction of dynamic iterations present in the *weak grouped* case in Figure 5.23. *Weak - no exact match* means that no exact matching is found for the loops, and *exact match* means that the loop is linked to another loop by an exact match.

Two programs, *sc* and *go*, greatly benefit from exact match groups. Most of *sc*'s exact match win comes from the iteration of rows and columns when performing mathematical computations on the spreadsheet. Figure 5.14(a) shows the code from the function, *doavg*. In this function, the loop iterating over the rows has an *init* of `minr` that is passed in as a parameter, and the

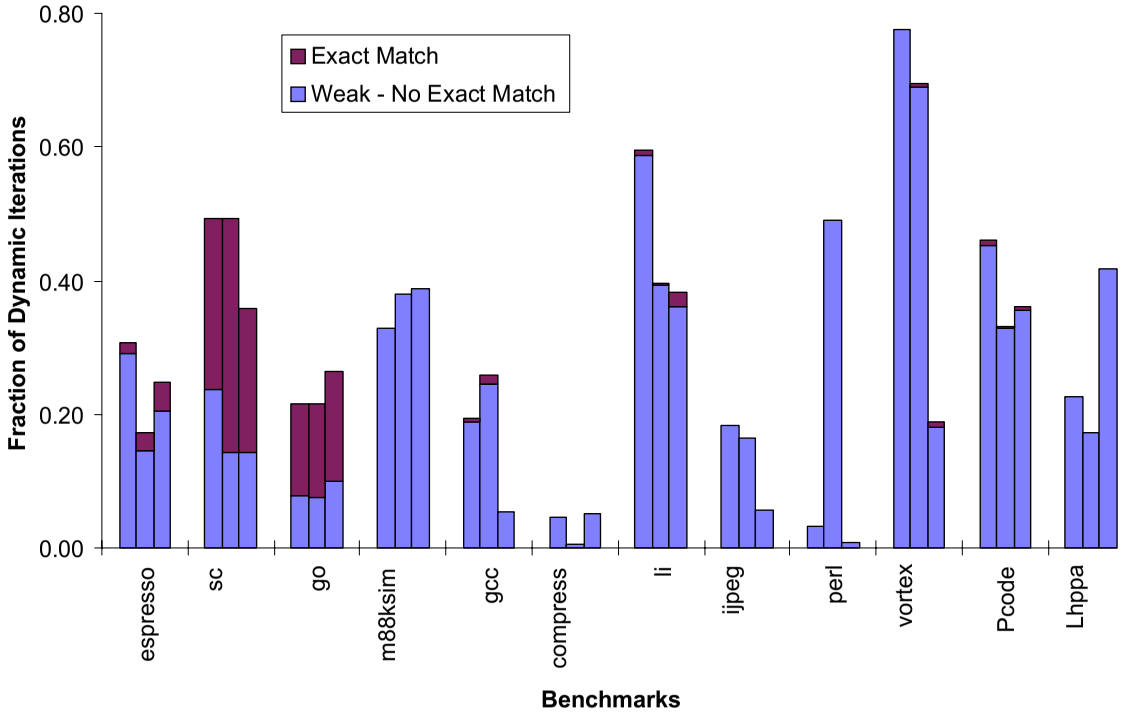

Figure 5.29 Applicability of exact match group functionality.

loop iterating over the columns has an *init* of `minc` that is also passed in as a parameter. All the spreadsheet computation functions, listed in the `switch` statement found in Figure 5.14(b), share these same *init*s and can be considered members of the same exact match group.

The benchmark *go* is helped similarly by exact match groups. The biggest win comes for forced-weak-grouping counted loops controlled by the global `ldir`. The global `ldir` is set to a constant with a size defined by the size of the game board. In the source code 75% of the loops controlled by `ldir` have the same *init* value, which is the global `fdir`. The global `fdir` contains the distance of a square from the edge of the board. The loops iterating with this loop control have consistent behavior, and much better loop-trip-count assertions based on loop grouping can be made.

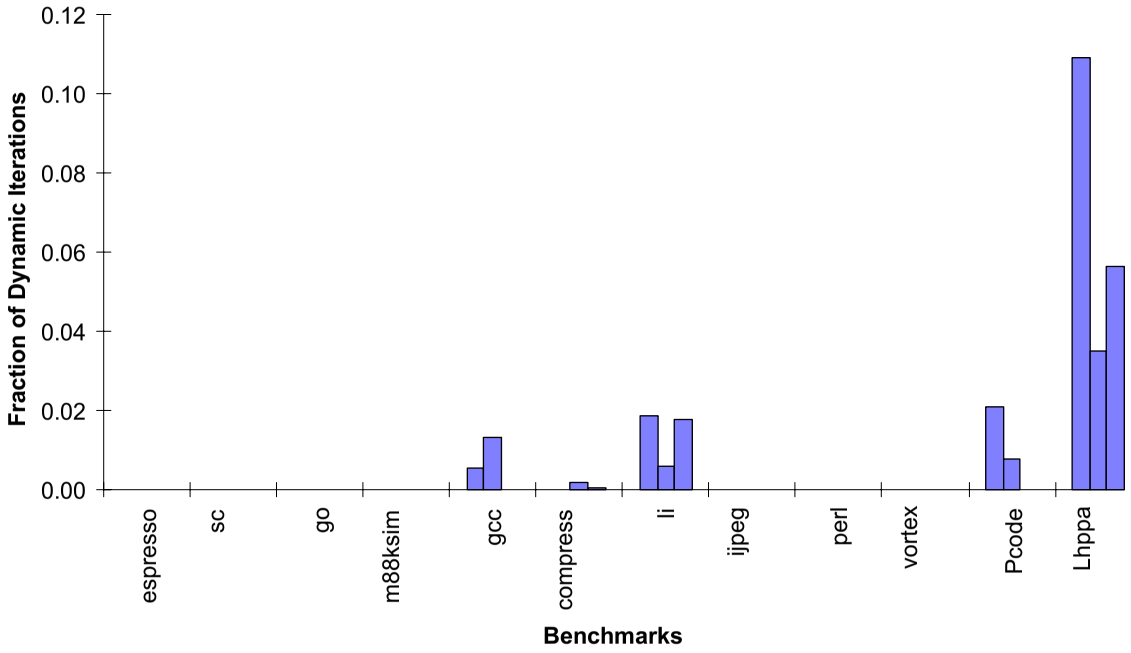

Figure 5.30 Applicability of equivalent group functionality.

5.3.4.3 Equivalent groups

Equivalent groups are useful because they allow loop groups that behave similarly to be merged. This allows loops that might be members of a small loop group to be combined with a much larger loop group. The equivalent groups' functionality is discussed in Section 5.2.4.3.

Figure 5.30 shows the importance of the loop groups involved in the equivalent groups. The bar in the figure represents the fraction of dynamic iterations for the loop groups involved in the merging of loop groups with the equivalent groups functionality.

The benchmark *li* is helped by the equivalent groups functionality. The equivalent group situation in *li* is discussed in Section 5.2.4.3. The equivalent group merging allows three loop groups, which would have been kept separate otherwise, to be grouped together. While they do not account for a large percentage of the dynamically important loops, they do account for 38% of the loops found in the source code.

Three other benchmarks, *gcc*, *Pcode*, and *Lhppa*, also have equivalent group situations. In the cases found in these benchmarks, loop groups with a small number of members are able to be merged with a loop group containing a much larger number of members. This is beneficial to the loops that would have been contained in a loop group with a small number of loops. However, the number loops, measured both by the number of loops in the source code and the number of dynamically important loops, helped in these benchmarks is small.

5.4 Potential Applications

The use of the loop grouping technology in estimating loop trip counts in loops that are unexercised during profiling is discussed in Chapter 6. However, there are other potential applications of this technology beyond its use in this dissertation. An extension to this technology could be used to predict unexercised, non-loop branches. Non-loop branches associated with globals and structure fields would be grouped, inferences on the values found in the globals and structure fields would be made based on the behavior of exercised branches, and predictions for unexercised branches would be made. These non-loop branches may also be difficult for the non-loop branch heuristics discussed in Section 4.2 to predict because these branches may not be located in code sections that contain the characteristics needed for predicting branch direction.

Loop grouping could also be used to override the trip count observed from profiling when a loop group exhibits inconsistent loop-trip-count behavior. The overriding would be done on loops that behave one way for an input, but have a good likelihood of behaving differently for another input. An example of where this idea would work well is presented in Section 5.3.3 and illustrated in Figure 5.27.

Loop grouping could also be used to provide feedback to the user who profiles a program. When users profile, they can benefit from knowledge of which functions are exercised. They can also benefit from knowledge about which functionality is unexercised, independent of whether the functions that contain the functionality are exercised. Loop grouping can accomplish this goal by monitoring when none of the loops contained in a loop group are exercised and reporting this fact to the user as a warning. The user can then decide if this functionality is important, and reprofile with a different input if necessary.

Another use of loop grouping could be to provide feedback during the process of fixing software bugs. If a bug is fixed in one loop, the other loops contained in the same loop group could be evaluated to see if the fix should be applied in other code locations. In this way, a bug that is located in multiple locations in the source code could be detected and fixed more easily.

CHAPTER 6

COMPILATION OF CODE UNTOUCHED DURING PROFILING

This chapter investigates the compilation of code that is unexercised during program profiling. As shown in Section 3.2, it is possible for important code segments of real programs to be unexercised during profiling. Because of this issue, a profile-based ILP compiler should spend a small part of its compile-time budget performing a few important optimizations to code that is unexercised during profiling. A small time investment at compile time can yield a big win at run time if the original profiling inputs do not capture all possible executions. The work in this chapter enables optimizations to be applied intelligently by predicting loop trip counts for loops that are unexercised during profiling. The work builds upon the loop grouping technology that is introduced in Chapter 5.

The static loop-trip-count prediction problem is difficult to solve. A detailed description of the issues involved with static loop-trip-count prediction is presented in Section 4.3. However, loop-trip-count prediction can be successfully performed on a large number of counted and linked list loops when these loops are unexercised during program profiling. This is possible by observing the trip counts of loops that are exercised during profiling, and using these observations to predict the trip count behavior of unexercised loops.

This chapter consists of the following sections. Section 6.1 provides an introduction to the potential problem and its solution. Section 6.2 discusses the actual methods for loop-trip-count

prediction. Section 6.3 presents the results of the prediction. Finally, Section 6.4 concludes with a brief review of the chapter and a discussion of future work.

6.1 Introduction

Code that is unexercised during program profiling can be important during real executions. Several factors account for the fact that not all the important command line options are used, or that not all the representative inputs are applied. This can occur through carelessness during the profiling process or an oversight during the development of the inputs for profiling. This can also occur when certain program executions take an extremely long time to finish. In this situation, users may not be willing or able to apply all the representative inputs. This problem is compounded for large programs that contain rich sets of functionality.

The potential problem is illustrated in Section 3.2. Figure 3.3 in that section shows the percentage of time that is spent executing code that is unexercised by one input, but used by another input. Five benchmarks have at least one input that spends more than 30% of its execution time in code that is unexercised by another input. While this situation should be avoided whenever possible by applying more profile inputs, it is still a significant issue that a production compiler must consider dealing with.

Not optimizing unexercised code can also limit the potential speedup possible with a strong optimizing compiler. If a compiler is good at extracting ILP from code that is exercised during profiling, but unable to do the same for code that is unexercised during profiling, the overall speedup for specific runs of an application will be limited by the percentage of code that is unexercised during profiling. This can lead to a situation for compiler optimization that is similar to Amdahl's law in computing.

In order to optimize unexercised code, the compiler needs guidance. One option is for the compiler to be guided by the same heuristics described in Chapter 4. In that chapter, static branch prediction and static loop-trip-count heuristics allow the compiler to infer information normally obtained from profiling. Unfortunately, the success of these heuristics, especially loop-trip-count prediction, is limited.

However, there is a major difference between the no-profile case described in Chapter 4 and the unexercised code case described in this chapter: profiling has been performed in the unexercised code case. Because profiling has been performed, data have been applied to the program. Therefore, another option is for the compiler to observe the behavior of the code that is exercised during profiling and use the information obtained from the exercised code to predict the behavior of the unexercised code. The remainder of this chapter uses this concept to allow the static prediction of loop trip counts in code that is unexercised during program profiling.

6.2 Loop Trip Count Prediction

Loop-trip-count prediction is performed on loops that are unexercised during program profiling. The predictions are made by using the loop grouping technology defined in the previous chapter and by inferring the behavior of unexercised loop group members from the behavior of exercised loop group members. The loop-trip-count categories used for prediction — low, medium, and high — are the same as those defined in Table 3.4.

Predictions are easily made when at least one strong set member of a loop group is exercised. The behavior of exercised strong loop group members is used to infer the behavior of all the members of the entire loop group. Unexercised strong loop group members are assumed to have

Table 6.1 Assertions made for weak group members when strong group members are exercised.

	Prediction for Weak Type	
Strong Trip Count	OR	AND
Low	–	Low
Low-Medium	–	Low-Medium
Medium	Medium-High	Low-Medium
Medium-High	Medium-High	–
High	High	–

the same trip count as the exercised strong loop group members. These predictions can span multiple categories (e.g., low-medium and medium-high) when at least one exercised strong loop group member has an average trip count in one category and another exercised strong loop group member has an average trip count in an adjacent category. While these predictions are not as *precise* (a more precise prediction means that the prediction encompasses fewer loop-trip-count categories — e.g., a low prediction is more precise than a low-medium prediction) as predictions that include just one loop-trip-count category, the multiple category predictions can be used to narrow down the possible optimizations that should be applied by a compiler. In addition, there are some cases where one loop has an average trip count just below a category cutoff (e.g., 4.9 is just below the low trip count cutoff of 5) and another loop has an average trip count just above a category cutoff (e.g., 5.1). Allowing multiple category predictions permits these cases to be counted as a prediction in spite of the arbitrary boundary definitions used in this dissertation.

The behavior of unexercised weak loop group members is a little more complicated to predict because the observed loop-trip-count behavior only provides a limit on the expected behavior of these loops. Table 6.1 shows the assertions that are possible for weak loop group members.

Table 6.2 Assertions made for strong group members when only weak group members are exercised.

Weak Type	Weak Trip Count	Prediction for Strong
OR	Low	Low
OR	Low-Medium	Low-Medium
OR	Medium	Low-Medium
AND	Medium	Medium-High
AND	Medium-High	Medium-High
AND	High	High

The loop-trip-count prediction of weak loop group members is limited because they are related to the strong loop group members by an upper bound for an AND relationship or lower bound for an OR relationship. Therefore, an inference is only possible for an AND relationship when the exercised strong loop group members have average trip counts less than or equal to medium. For an OR relationship, exercised strong loop group members must have average trip counts higher than or equal to medium. The dashes in Table 6.1 represent cases where a confident loop-trip-count prediction for the weak loop group member is not possible. Section 6.2.3 discusses the possibilities of less confident predictions for the weak loop group member cases represented by dashes in Table 6.1.

There are instances where none of the strong loop group members are exercised during program profiling, but at least one of the weak loop group members is exercised. In this case, assertions about the expected loop trip count of the strong loop group member can sometimes be made. The assertions are possible because the weak loop group members are related to the strong-loop-group-member trip count by a lower or upper bound, and the observed trip count for a weak loop group member permits the strong-loop-group-member trip count to be determined. Table 6.2 shows the cases where an exercised weak loop group member can predict the trip count for a strong loop group member.

The remainder of Section 6.2 provides the details of how loop trip counts of unexercised loops are predicted. Section 6.2.1 provides details on the data structures used during the prediction process. Section 6.2.2 describes the actual algorithm used for loop-trip-count prediction. Section 6.2.3 provides a means of predicting more weak group members, but with a lower confidence in the prediction being correct. The success of these loop-trip-count predictions is evaluated in Section 6.3.

6.2.1 Data structures

There are three major data structures that are used to predict loop trip counts for unexercised loops. They include loop groups, subgroups, and exact match groups. The loop group and subgroup data structures are created in the counted loop and linked list loop detection algorithms discussed in Sections 5.2.2.3 and 5.2.3.3. This is done in the "loop group branch" lines of each algorithm. The exact match group data structures are created after all the loop groups and subgroups have been determined. This is accomplished by evaluating all the weak correlation set members of the loop group or subgroup as described in Section 5.2.4.2.

Loop groups are the major data structures used in loop grouping and loop-trip-count prediction. They correspond to the counted and linked list loop groups described in Chapter 5. Each individual loop group corresponds to a single loop control mechanism (e.g., a global-controlled counted loop for a particular global variable, or a function-controlled linked list loop for a particular function). The loop group data structure contains a list of the loops that are strongly correlated to the loop grouping mechanism, a list of loops that are weakly correlated to the loop grouping mechanism, and a list of exact match groups that can be formed from the weakly

correlated loop list. In addition, for a linked list loop group, the list of subgroups for the loop group members is maintained.

Subgroups are used to handle the problem of generic linked lists, and they are described in Section 5.2.4.1. Subgroups are formed for linked list loop groups, and the list of subgroups for a linked list loop group is kept in the loop group data structure. Each subgroup contains three lists: one for its strongly correlated set members, one for its weakly correlated set members, and one for its exact match group members.

Exact match groups are used to link weak correlation set members of subgroups and loop groups that share the exact same loop control. The advantage of this grouping is that all the exact match loops are strongly correlated to one another, and this fact can be leveraged when making loop-trip-count assertions. Section 5.2.4.2 covers the exact match functionality. A list of all the exact match groups for each loop group and subgroup is kept in the parent loop group or subgroup. The exact match group contains one list that contains all the loops that share the exact same loop control. By definition, this list only contains loops that are strongly correlated to one another.

6.2.2 Algorithm

The algorithm for predicting loop trip counts of unexercised loops is composed of two major parts. The first half of the algorithm determines the inferred minimum and maximum trip counts based on the behavior of the loops that are exercised, and computes a prediction for unexercised loops based on the minimum and maximum average trip counts that are observed. The second half of the algorithm uses these predictions to determine the predicted trip count for unexercised loops.

```
loop_group_predict (loop_group)
{
    subgroup_predict (loop_group->first_subgroup)
    exact_match_predict (loop_group->first_exact_match)
    compute_pred (loop_group)
    determine_actual_predictions (loop_group)
}

subgroup_predict (first_subgroup)
{
    For first_subgroup and each subgroup sibling of first_subgroup {
        exact_match_predict (subgroup->first_exact_match)
        compute_pred (subgroup)
    }
}

exact_match_predict (first_exact_match)
{
    For first_exact_match and each exact_match sibling of first_exact_match {
        compute_pred (exact_match)
    }
}

compute_pred (group)
{
    Determine min and max average trip counts for strong members of group
    If (no exercised strong members of group)
        Infer min and max average trip counts based on weak members of group
    Compute prediction for group based on min and max values
}
```

Figure 6.1 High-level algorithm for predicting loop trip counts.

The high-level algorithm for predicting loop trip counts is shown in Figure 6.1. The function *loop_group_predict* is called once for every loop group found in the program. The first three function calls in *loop_group_predict* perform the functionality of the first half of the loop-trip-count prediction algorithm — computing the prediction that should be used for unexercised loops — and the fourth function call to *determine_actual_prediction* initiates the second half of the prediction algorithm — actually predicting the trip counts for unexercised loops.

The algorithm is broken down into several functions in order to simplify the traversal of the data structures associated with loop grouping. The function *loop_group_predict* determines the prediction to be used for unexercised loops based on the loop group, and it initiates the

second half of the loop-trip-count prediction algorithm. The function *subgroup_predict* determines the predictions to be used for the loops contained in the subgroups. The function *exact_match_predict* determines the predictions to be used for the loops in the exact match groups. Finally, *compute_pred* determines the minimum and maximum average trip counts that occur in exercised loops, and computes the prediction that should be applied to unexercised loops. The function *compute_pred* is a common function that is used by all three functions associated with the different data structures.

The top-level function, *loop_group_predict*, is visited once for every loop group that is found in the program. The function initiates the determination of predictions for the subgroups and exact match groups in the first two lines of the function. The loop-trip-count predictions that should be applied to unexercised loop groups is determined in the call to *compute_pred*. The predictions that are determined for the loop group as a whole are applied if a subgroup or exact match group is unable to provide a prediction. The mechanism for prediction of unexercised loops is described later in this section.

The subgroup prediction function, *subgroup_predict*, is similar to *loop_group_predict*. The function iterates over all the subgroup siblings, and calls *exact_match_predict* and *compute_pred* to determine the predictions that should be used. The exact match prediction function, *exact_match_predict*, visits all the exact match siblings formed from the same loop group or subgroup. It only needs to call *compute_pred* because it does not contain any other data structures that require further processing.

The *compute_pred* function determines the minimum and maximum average trip counts present for the loops in the incoming data structure, termed *group*, whether it is a loop group, subgroup, or exact match group, and uses the minimum and maximum information to determine

a loop-trip-count prediction that should be applied to unexercised loops. The minimum and maximum average loop trip counts are determined from the averages observed in the exercised strong correlation set members of the group currently being analyzed. If no strong correlation set members are exercised, the weak correlation set members are analyzed. Table 6.2 shows the assertions that can be made for the strong correlation set members when only weak correlation set members are exercised. Finally, the predictions that should be applied to unexercised loops are determined based on the minimum and maximum values. Where the minimum and maximum values fall in Table 3.4 dictates the prediction category for unexercised loops.

After all the minimum, maximum, and prediction information has been determined for each data structure, the actual loop-trip-count prediction for unexercised loops is performed. This is done in the function *determine_actual_predictions*, which is presented in Figure 6.2. This function is called from *loop_group_predict* in Figure 6.1.

The algorithm for determining the actual loop-trip-count predictions is broken down into three functions for easy traversal of the data structures. The order of traversal is important in determining a precise prediction for a loop. This means that the exact match groups are visited first, and subgroups are visited before the loop group. The function *determine_actual_prediction* performs the prediction for unexercised loops in the loop group. The function *determine_actual_pred_for_subgroup* performs the prediction for unexercised loops in the subgroups. Finally, *determine_actual_pred_for_exact_match* predicts all the unexercised loops in the exact match groups.

The function *determine_actual_prediction* initiates the process of determining predictions for unexercised loops. It starts by allowing all the subgroups and exact match groups to predict the unexercised loops. Because the loop group is a superset of all the subgroups and exact

```
determine_actual_predictions (loop_group)
{
    determine_actual_pred_for_subgroup (loop_group->first_subgroup)
    determine_actual_pred_for_exact_match (loop_group->first_exact_match)
    If (loop_group->pred != NONE) {
        For each strong loop member of loop_group {
            If (strong is unexercised && strong->pred != NONE)
                strong->pred = loop_group->pred
        }
        For each weak loop member of loop_group {
            If (weak is unexercised && weak was not predicted by a subgroup or exact_match)
                weak->pred = update_weak_prediction (loop_group->pred, weak->pred)
        }
    }
}

determine_actual_pred_for_subgroup (first_subgroup)
{
    For first_subgroup and each subgroup sibling of first_subgroup {
        determine_actual_pred_for_exact_match (subgroup->first_exact_match)
        If (subgroup->pred != NONE) {
            For each strong loop member of subgroup {
                If (strong is unexercised && strong->pred != NONE)
                    strong->pred = subgroup->pred
            }
            For each weak loop member of subgroup {
                If (weak is unexercised && weak was not predicted by a subgroup or exact_match) {
                    weak->pred = update_weak_prediction (subgroup->pred, weak->pred)
                    Mark weak as predicted by a subgroup
                }
            }
        }
    }
}

determine_actual_pred_for_exact_match (first_exact_match)
{
    For first_exact_match and each exact_match sibling of first_exact_match {
        If (exact_match->pred != NONE) {
            For each strong loop member of exact_match {
                If (strong is unexercised) {
                    strong->pred = exact_match->pred
                    Mark strong as predicted by an exact_match
                }
            }
        }
    }
}
```

Figure 6.2 Algorithm for determining actual loop-trip-count predictions.

match groups, the loop group will never be able to make a more precise prediction than any subgroup or loop group. The loop group is used to predict unexercised loops that are not part of any subgroup or exact match group.

After all subgroup and exact match group predictions are applied, the prediction for unexercised, unpredicted loop group members is performed, assuming a prediction is possible. A loop group that has no strong or weak loop group members that are exercised cannot have any predictions made because exercised loop group members are necessary for predictions to be made. If this situation arises, there is no prediction for any of the loops in the loop group (*loop_group–>pred* equals NONE).

When a prediction is possible, every strong loop group member loop that is unexercised and has not already been predicted is given the prediction associated with the loop group. Every weak group member loop that is unexercised and not previously predicted by a subgroup or exact match group in the current loop group is predicted using the *update_weak_prediction* function described later in this section.

The function *determine_actual_pred_for_subgroup* is similar to *determine_actual_prediction* in Figure 6.2, except the subgroup's fields are accessed and all the subgroup siblings are traversed. The function *determine_actual_pred_for_exact_match* is also similar to *determine_actual_prediction*, except that it is simpler because an exact match group only contains a list of strongly correlated members.

The function *update_weak_prediction* is shown in Figure 6.3(a). This function is called from the functions *determine_actual_prediction* and *determine_actual_pred_for_subgroup*, and is used to determine the prediction for a weakly correlated loop group member. Table 6.1 details the loop-trip-count assertions that are possible for weak correlated loop group members.

```
update_weak_prediction (curr_pred, old_pred)
{
        new_pred = old_pred
    if (AND relationship || forced weak group loop) {
        if (curr_pred == LOW)
            new_pred = infer_prediction (old_pred, LOW)
        else if (curr_pred == LOW-MEDIUM || curr_pred == MEDIUM)
            new_pred = infer_prediction (old_pred, LOW-MEDIUM)
    }
    else { /* OR relationship */
        if (curr_pred == HIGH)
            new_pred = infer_prediction (old_pred, HIGH)
        else if (curr_pred == MEDIUM-HIGH || curr_pred == MEDIUM)
            new_pred = infer_prediction (old_pred, MEDIUM-HIGH)
    }
    return (new_pred)
}
```

(a)

old_pred \ grp_pred	None	L	LM	M	MH	H
None	N/A	L	LM	M	MH	H
L	N/A	L	L	Err	Err	Err
LM	N/A	L	LM	M	M	Err
M	N/A	Err	M	M	M	Err
MH	N/A	Err	M	M	MH	H
H	N/A	Err	Err	Err	H	H

LEGEND
L = Low
LM = Low-Medium
M = Medium
MH = Medium-High
H = High
Err = Error

(b)

Figure 6.3 Algorithms for determining weak predictions: (a) update_weak_pred algorithm, (b) prediction table for infer_prediction.

The function *update_weak_prediction* has two parts. One deals with the AND relationship and forced-weak-group conditions that allow an upper bound on the expected loop trip count to be made. The other deals with the OR relationship that allows a lower bound on the expected loop trip count to be made. The conditions inside each of these sections correspond to the entries in Table 6.1 that do not contain dashes.

The function *update_weak_prediction* uses the prediction for an unexercised loop based on the current loop group or subgroup, *curr_pred*, and the old prediction for the unexercised loop,

old_pred, as its inputs, and outputs the new prediction for the unexercised loops, *new_pred*. The old and new predictions are used because multiple loop group categories can apply to the same loop. The predictions need to account for each loop group because one may be more restrictive than the other, and the most precise prediction for the unexercised loop is desired.

The function *infer_prediction* is used to resolve the case when multiple loop groups apply to the same unexercised loop. The table located in of Figure 6.3(b) shows the functionality of *infer_prediction*. The left column of the table, *old_pred*, corresponds to the first parameter passed into the function, and the top row of the table, *grp_pred*, corresponds to the second parameter passed in the function. The entry of a particular row and column corresponds to the function return value. This function returns the more restrictive prediction. For example, if the *old_pred* is low-medium and *grp_pred* is low, the returned prediction is low. If there is no prior prediction for a loop, *old_pred* is assumed to have a value of none. An error entry means that an impossible situation is detected.

The functions shown in Figures 6.1, 6.2, and 6.3 represent the entire loop-trip-count prediction algorithms needed to predict loop trip counts for unexercised loops. There is good confidence that the predictions made through this analysis are accurate. The success of these techniques to predict unexercised loop trip counts is examined in Section 6.3. An analysis of some other predictions that can be made, but are less accurate, is presented in the next section.

6.2.3 Weaker confidence predictions

This section describes extensions to the work presented in the previous section so that more loop-trip-count predictions can be made. This is preliminary work that illustrates what can be done to produce predictions for the cases represented by dashes in Table 6.1. A more sophisti-

cated method of dealing with these cases is needed in order for the weaker confidence predictions to be more useful. The effectiveness of this preliminary work is covered in Section 6.3.3.

The assumption made in this preliminary work is that the trip count for the loops that are represented by dashes in Table 6.1 matches the expected strong correlation trip count. This prediction is only made if no prediction described in the previous section is applied. The idea for this prediction is that there is at least one condition in the loop that exhibits the strong correlation trip count, and there are no conditions in the loop that contradict the strong correlation trip count. Because of these facts, there is a greater likelihood that the loop will iterate a similar number of times to the strong correlation trip count.

The effectiveness of this assumption is measured in Figure 6.4 by analyzing how well it works for weak correlation loops that are exercised by the different benchmarks and inputs. In this figure, the x-axis shows the benchmarks, and the bars for each benchmark represent the results for the three inputs that get applied. The y-axis is the fraction of dynamic loop iterations, which are defined and used in Section 3.2. The height of each bar represents the fraction of the total dynamic loop iterations for the input that would fit into a case represented by a dash in Table 6.1. This experiment assumes that the exercised loops would be predicted using the prediction assumption described in the previous paragraph. Assuming these loops are predicted using this assumption, each bar is further broken down into two categories: *pred correct* and *pred incorrect*. A loop is considered correctly predicted, *pred correct*, if its loop trip count matches the trip count of the expected strong trip count. Otherwise, the loop is considered to be incorrectly predicted, *pred incorrect*. A correct prediction means that if an exercised loop needed to be predicted, the prediction would have been based on the work in this section, and the prediction would have been correct.

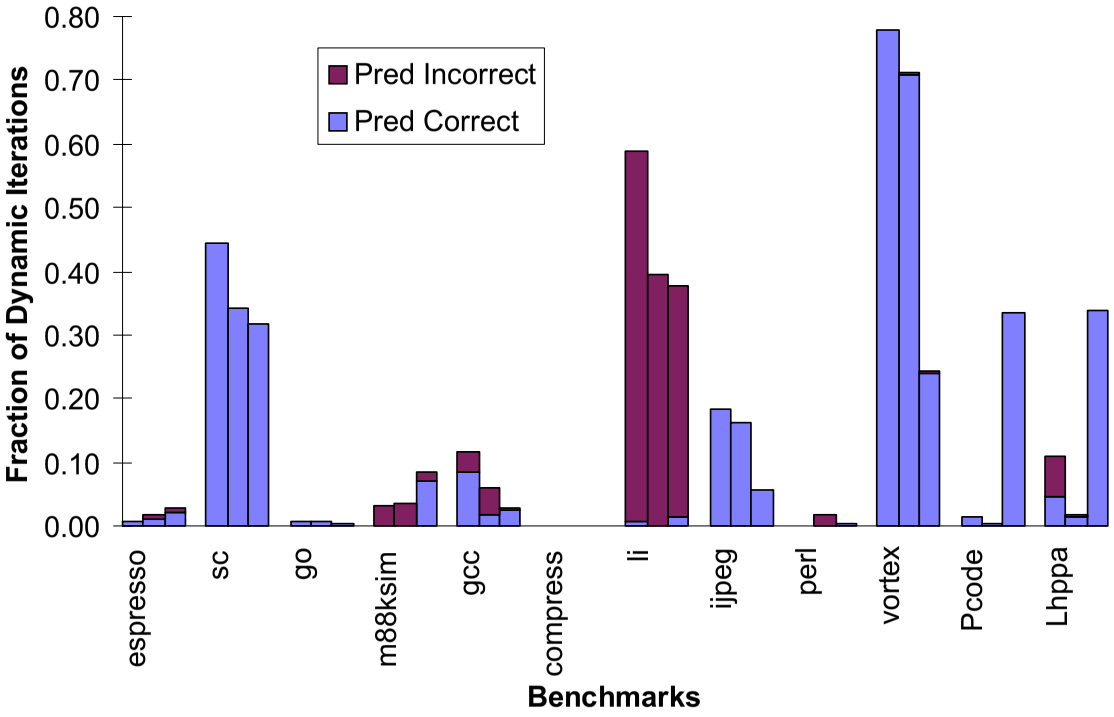

Figure 6.4 Potential opportunity for correct prediction of more weak loop group members.

As seen in Figure 6.4, the majority of the weak correlation loops in the benchmarks would have been correctly predicted. The major exception occurs in the benchmark *li*. An enhancement to the preliminary work done in this section would be to not always use the expected strong correlation trip counts. If other weak correlation set members are exercised and fall into one of the hard-to-predict cases, the unexercised weak correlation set members should have predictions matching the exercised weak correlation set members. This holds promise because hard-to-predict loops seem to act similarly for the same benchmark.

Another potential enhancement to this preliminary work is to consider the number of conditions that must be true in order for a loop to continue iterating. For example, a loop that contains 10 branches having an AND relationship will probably iterate fewer times than a loop that only contains 2 branches having an AND relationship.

The weaker confidence predictions are done after all the predictions described in the previous section have been made. In this way, weak confidence predictions are only going to be applied to loops that would not be predicted otherwise. The algorithm for prediction is simple. If a loop is a member of a weakly correlated set in a loop group, if it is unpredicted, and if the loop prediction falls into a category represented by a dash in Table 6.1, then predict that the trip count matches the expected trip count of the strongly correlated set members in the loop group.

6.3 Loop-Trip-Count Prediction Results

This section evaluates the effectiveness of loop grouping and the loop-trip-count prediction techniques described in this chapter to predict unexercised loops. This analysis is performed on Lcode files that have gone through a minimal compilation path. For these Lcode files, the only optimizations that are applied are IMPACT's classical optimizations. After optimization, the code is profiled. The results of profiling are used to facilitate the measurement of success for the static loop-trip-count prediction.

This section is composed of three subsections. Section 6.3.1 evaluates the ability of the techniques to predict trip counts for unexercised loops. Section 6.3.2 evaluates the effectiveness of trip count prediction for loops unexercised by one input but exercised by another input. Finally, Section 6.3.3 looks into the effectiveness of the weak confidence prediction heuristics.

6.3.1 Unexercised loops

The first experiment evaluates how well that the loop grouping techniques described in Section 6.2.2 are able to predict loops that are not exercised by a particular benchmark-input

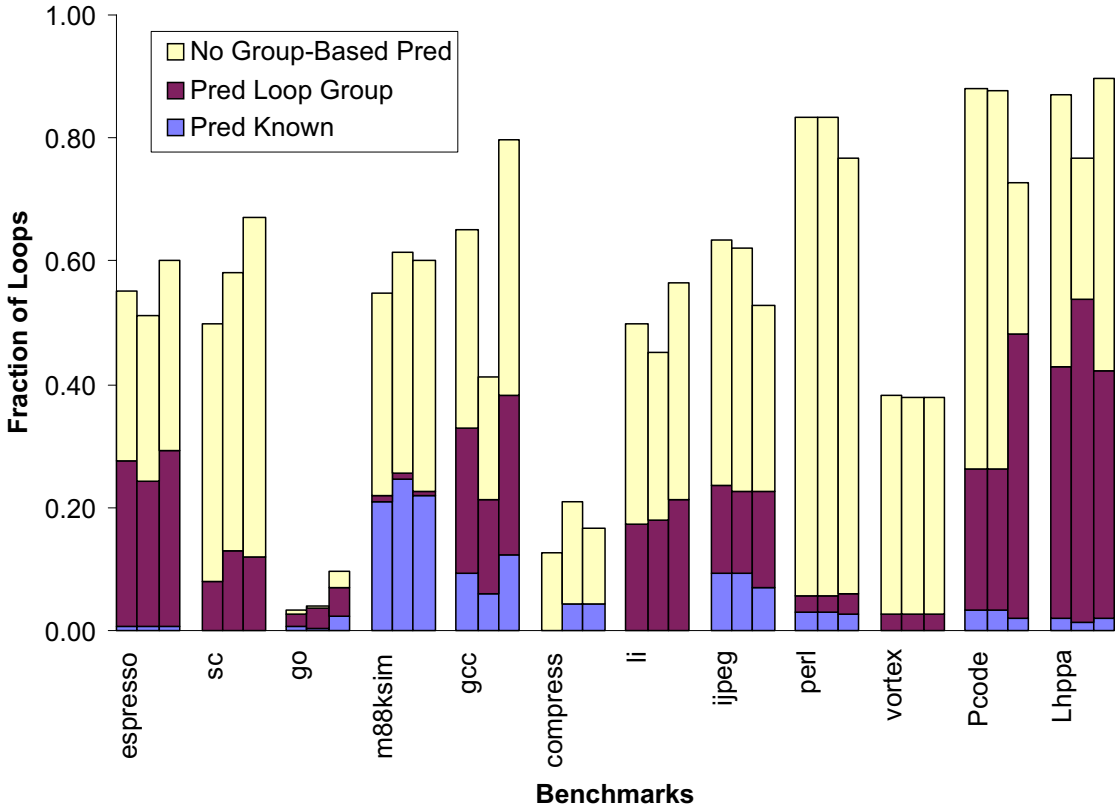

Figure 6.5 Loop-trip-count prediction for unexercised loops.

pair. The results of prediction for all the unexercised loops in each benchmark are shown in Figure 6.5. The x-axis contains the benchmarks and inputs used in this analysis. The first bar for each benchmark represents the behavior seen when input 1 is applied, the middle bar represents the behavior when input 2 is applied, and the third bar represents the behavior when input 3 is applied. The y-axis is the fraction of all the loops contained in the source code, which is a static measure instead of the dynamic measure used in most of the results of this dissertation. The height of each bar represents the fraction of loops in the source code that are unexercised by the benchmark's input.

Each bar is composed of three parts. *Pred known* means that a prediction is possible for the loops because the loop is controlled by constants. These loops can be predicted with

technology present in current compilers. *Pred loop group* means that a prediction is possible using the algorithms presented in Section 6.2.2. These predictions are only possible through the use of the loop grouping technology. Finally, *no group-based pred* means that no prediction is possible using the techniques described in Section 6.2.2. However, this does not mean that these loops cannot be predicted with enhancements to the loop grouping technique, like the weaker confidence predictions described in Section 6.2.3, or other predictions techniques, like the static loop-trip-count heuristics that are described in Section 4.3.

A significant number of unexercised loops are predicted with the loop grouping technology. However, there is still room for improvement. There are several reasons that the number of predicted loops is not higher. First, only "sure" predictions are being made. The weaker confidence prediction work described in Section 6.2.3 allows more predictions to be made. Second, only counted and linked list loops are predicted. Argument, file, and character processing loops described in Section 4.3 are not captured by the loop grouping technology, and can represent a significant number of loops. Heuristics can be developed that allow these loops to be identified, and trip count predictions would be possible for them. Third, for loop grouping to aid in the prediction of loop trip counts, some loop group members must be exercised, while other loop group members are unexercised.

There are several cases where the functionality associated with a large number of loops is unexercised. For example, in *Lhppa* the region functionality is not utilized by any input [46]. This functionality creates 4 loop groups that control the iteration of 41 loops. Because the functionality is unexercised, no predictions can be made for these loops. If the region functionality is important to potential users of the *Lhppa* program, it should be included in at least one of the inputs applied during profiling. Loop grouping helps unexercised loops that are similar

to other loops that have been exercised during profiling, but it does not help if no similar loops have been exercised during profiling.

As seen in the figure, there are many loops in the source code that are unexercised by one input. Only *go*, *compress*, and *vortex* have less than 40% of the loops in their source code unexercised by each of the inputs. The benchmarks, *gcc*, *perl*, *Pcode*, and *Lhppa*, have at least one input that exercises less than 20% of the loops.

The large number of unexercised loops can occur for several reasons. The major reason is that the programs that exercise less than 20% of the loops are large, complex programs. They contain a large amount of code to handle their varied functionality and handle many possible cases, and it is just not possible to exercise all of it with a single input.

Another reason is that there is dead functionality contained in these programs. Classical optimizations allow dead code inside of a function to be removed, but dead functions remain. For example, it has been shown that a little over 30% of the functions contained in *ijpeg* are dead [8]. The removal of these functions requires the resolution of all indirect function calls, and this analysis is not performed for this dissertation.

As illustrated by these results, future work is necessary before the predictions possible with loop grouping can be successfully used in a production compiler. The major item that needs to be determined is how to best spend the small part of the compile-time budget that should be dedicated to the optimization of unexercised code. This is a nontrivial problem to solve because of the high percentage of code that can be unexercised during profiling.

6.3.2 Unexercised input cases

The next experiment evaluates how well the unexercised loops for the cases shown in Figure 3.3 can be predicted. In Figure 3.3, an input for each benchmark is used to guide compilation, and then the other inputs are applied individually. This experiment measures how many of the loops unexercised by a profiling input and used by another input can be predicted.

Figure 6.6 shows the results of this prediction. The x-axis contains the benchmarks and contains one bar for each possible combination of profiling input and actual application input. The first bar has input 1 guiding compilation and input 2 being applied, the second bar has input 2 guiding compilation and input 1 being applied, the third bar has input 2 guiding compilation and input 3 being applied, the fourth bar has input 3 guiding compilation and input 2 being applied, the fifth bar has input 1 guiding compilation and input 3 being applied, and, finally, the sixth bar has input 3 guiding compilation and input 1 being applied. The y-axis is the fraction of dynamic iterations exercised. The height of each bar represents the fraction of dynamic iterations that are unexercised by the profiling input, but used by the input that is actually applied. The *pred known*, *pred loop group*, and *no group-based pred* categories have the same meanings they had for Figure 6.5.

Two benchmarks, *sc* and *ijpeg*, have almost all of the important, unexercised loops predicted using the loop grouping technology. For *sc*, the unexercised loops occur in the spreadsheet computations. Inputs 2 and 3 perform the sum computation on a set of cells in the spreadsheet, while input 1 performs all the computations except for sum. Loop grouping allows all the loops in the computation functions to be grouped together. For *ijpeg*, the data structures used for compression and decompression are shared. This allows the compression performed for inputs 1 and 2 to predict the behavior of loops only used for the decompression, which is performed

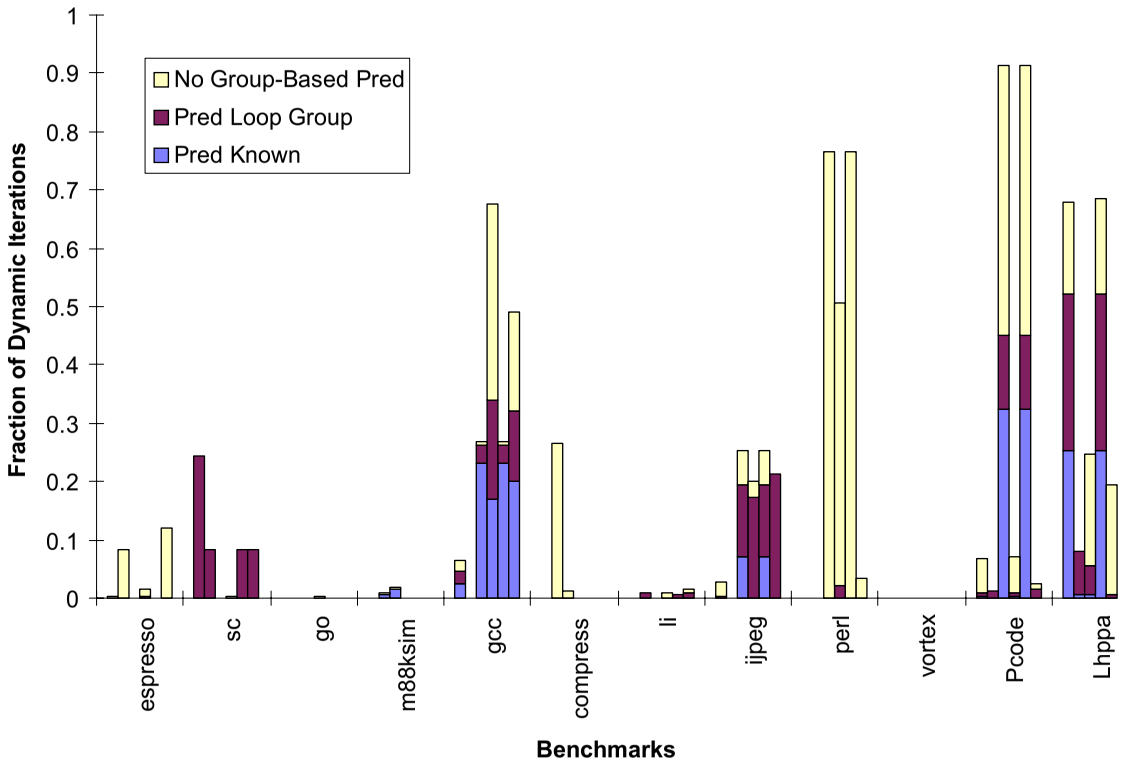

Figure 6.6 Loop-trip-count prediction for code unexercised by one input.

for input 3. The decompression input is also able to predict the behavior of loops only used for compression.

Two benchmarks, *compress* and *perl*, do not have many important, unexercised loops predicted. This occurs because these benchmarks are unable to utilize loop groups well. The issues associated with loop grouping in these two benchmarks are described in Section 5.3.2. Three benchmarks, *gcc*, *Pcode*, and *Lhppa*, are able to have a significant percentage of loops predicted, loops which are unexercised by one input and are used by another input.

Finally, the benchmark *espresso* does not have any important, unexercised loops predicted. However, these loops are contained in loop groups. Input 1 is the only input that exercises the binary variables functionality. The other two inputs do not exercise this functionality so the

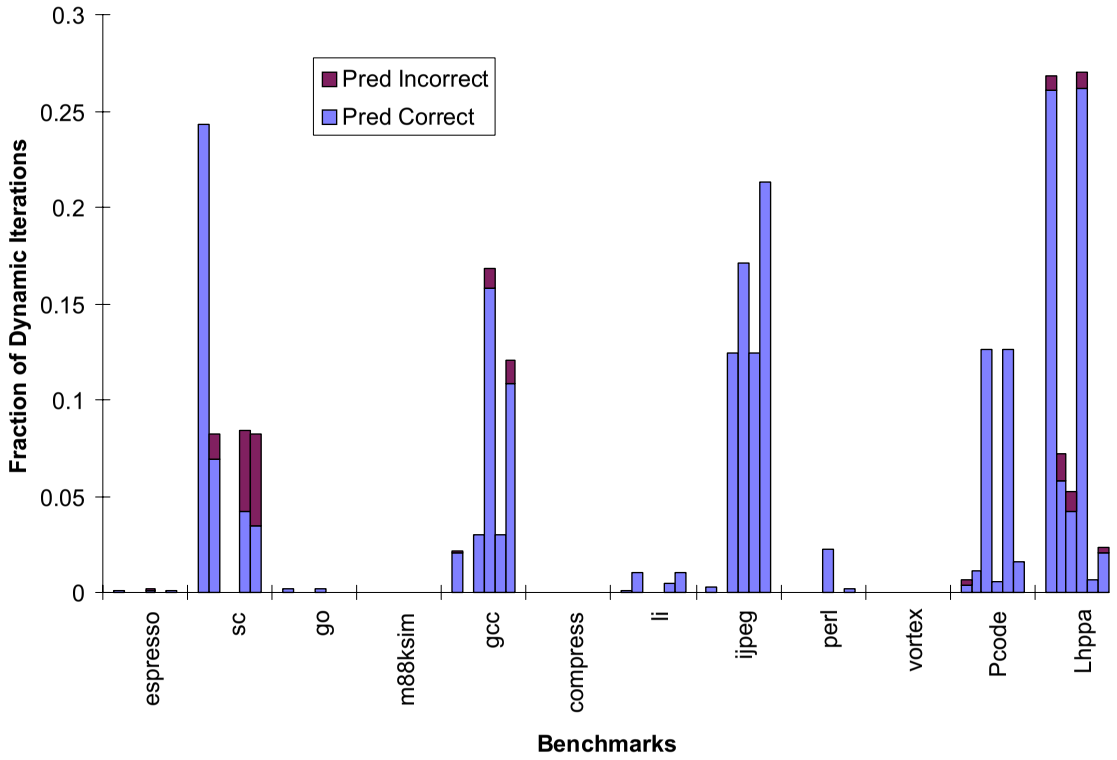

Figure 6.7 Accuracy of the loop-trip-count predictions.

loop groups for the binary variables do not have any exercised loop group members from which to obtain predictions.

Now the correctness of the predictions shown in Figure 6.6 is evaluated. The correctness of the predictions is shown in Figure 6.7. The x-axis and y-axis have the same definitions as they do in Figure 6.6. The height of each bar in Figure 6.7 represents the fraction of dynamic iterations spent in loops that are unexercised by the profiling input, used by the applied input, and can be predicted using the loop grouping technology. This corresponds to the *pred loop group* category in Figure 6.6.

For the loops predicted using the loop grouping technology, a correct prediction means that the actual trip count observed for the applied input is not in conflict with the prediction based

on the profiling input. For example, if a loop has an observed low average trip count and the prediction is low-medium, a correct prediction is assumed. However, if a loop has an observed low average trip count and the prediction is medium, an incorrect prediction is assumed.

As seen in the figure, the majority of the predictions made with the loop grouping technology are correct. The major exception occurs in the benchmark *sc*. In *sc*, the problem occurs because of the different number of rows that are accessed when performing computations on the spreadsheet. Input 3 performs a sum, and the average number of rows that are traversed is 180. Input 1 performs every computation but a sum, and the average number of rows that are traversed is never greater than 10. Therefore, the incorrect predictions are a function of the inputs that get applied. They do not necessarily represent a fundamental flaw in the use of loop grouping to predict trip counts.

6.3.3 Weaker confidence predictions

This section evaluates the effectiveness of the preliminary weaker confidence prediction work described in Section 6.2.3. The goal of the weaker confidence predictions is to predict cases that are unpredicted by the algorithms described in Section 6.2.2, but the cases contain loops that are grouped, and at least one group member is exercised. The predictions are performed on the cases corresponding to the dashes in Table 6.1. These predictions are possible, but the confidence that the correct predictions are being made is lower than for the predictions made in Section 6.2.2.

Figure 6.8 shows the fraction of the static loops that are predictable using the weaker confidence predictions. This figure is very similar to Figure 6.5. The difference is that the *pred known* and *pred loop group* categories from Figure 6.5 are combined into the *pred strong*

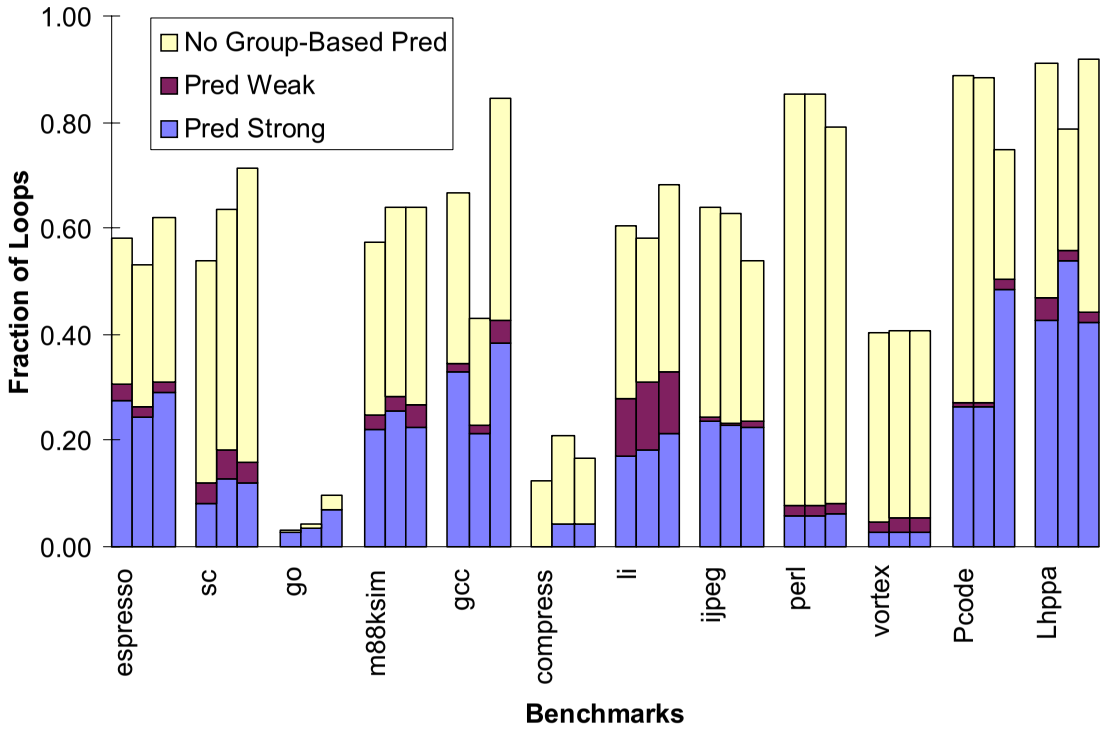

Figure 6.8 Weak confidence loop-trip-count prediction for unexercised loops.

category in Figure 6.8. In addition, the fraction of the loops that can have a weaker confidence prediction applied is shown in the *pred weak* category.

As seen in the figure, the weak confidence predictions allow more unexercised loops to be predicted. The benchmark benefitting the most from these predictions is *li*. For *li*, over 20% of the loops that are unexercised and unpredicted previously can now have predictions made for them.

The effectiveness of the weak confidence predictions for predicting loops using the inputs used in this dissertation is shown in Figure 6.9. This figure is very similar to Figure 6.6. The difference is that the *pred known* and *pred loop group* predictions made in Figure 6.6 are combined into the *pred strong* category; the fraction of loops that can have a weaker confidence

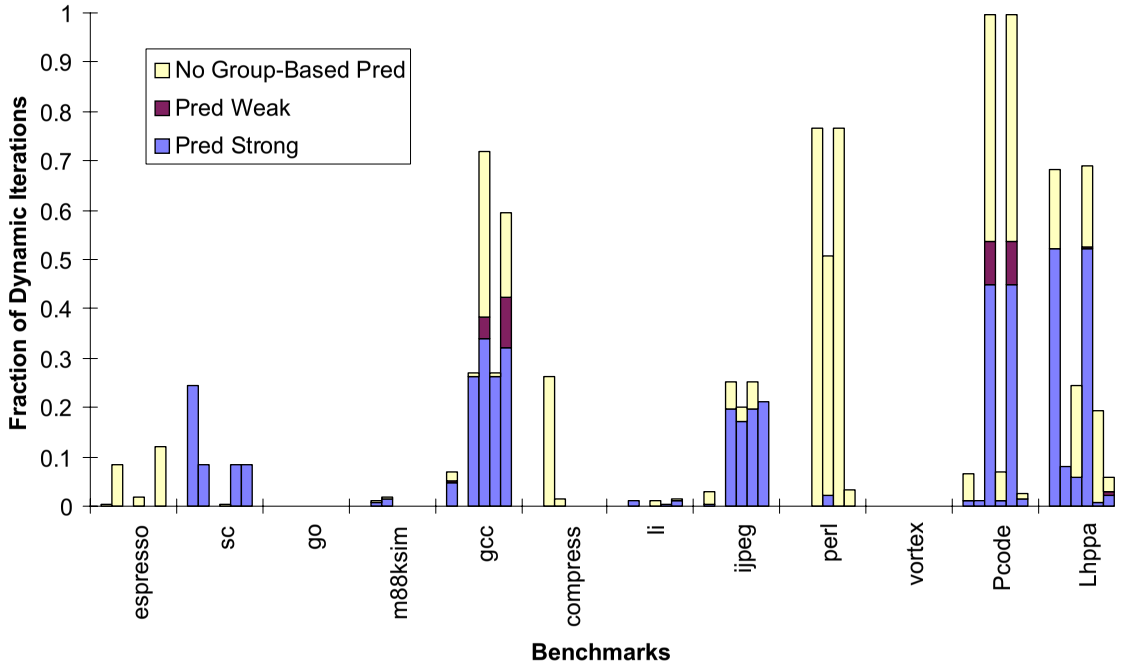

Figure 6.9 Weak confidence loop-trip-count prediction for code unexercised by one input.

prediction applied is shown in the *pred weak* category. The only two benchmarks that benefit significantly from the weaker confidence predictions in these situations are *gcc* and *Pcode*.

Now the correctness of the weaker confidence predictions shown in Figure 6.9 is evaluated. The correctness of the predictions is shown in Figure 6.10. The x-axis and y-axis have the same definitions as in Figure 6.9. The height of each bar in Figure 6.10 represents the fraction of dynamic iterations spent in loops that are unexercised by the profiling input, used in the applied input, and predicted using the weak confidence prediction — this corresponds to the *pred weak* category in Figure 6.9. The same definition of correctness that is used for Figure 6.7 is used in this figure.

As seen in the figure, *gcc* is not predicted well, while *Pcode* is predicted well. Most of the problems with *gcc* occur when input 3 is used for profiling and one of the other inputs is applied. It occurs because a single linked list loop group has one strong correlation set member

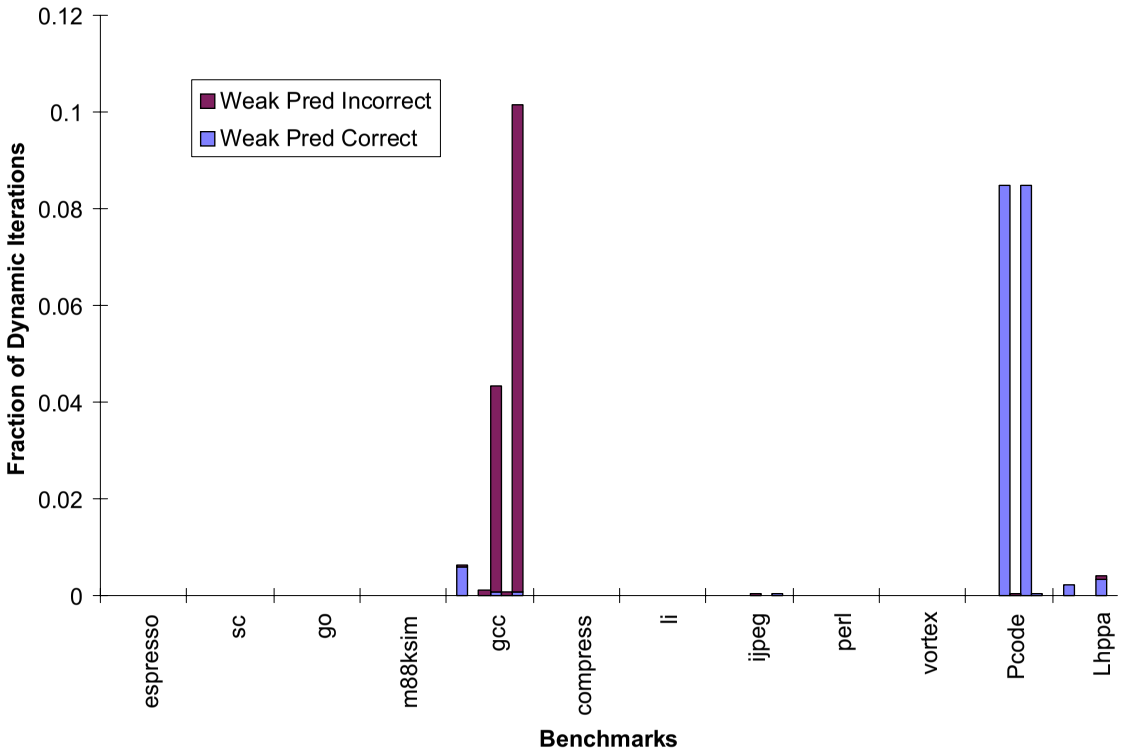

Figure 6.10 Accuracy of the weak confidence loop-trip-count predictions.

exercised in input 3. For input 3, the linked list shows a high trip count. However, when the other inputs are applied to *gcc*, this same strong correlation member is unexercised. Other strong correlation set members are exercised by the other inputs, and all of them have a short trip count. The incorrect predictions occur because of this inconsistency.

The weaker confidence predictions described in Section 6.2.3 are preliminary attempts at obtaining more loop-trip-count predictions. As seen in the results in this section and in the results obtained in Figure 6.4, more sophistication is needed in order for the weaker confidence predictions to be viable. Suggestions for possible enhancements are described in Section 6.2.3.

6.4 Conclusion

This chapter covers the prediction of trip counts for loops that are unexercised during profiling. This analysis is needed because it is possible for code that is important during the actual execution of a program to be unexercised during profiling. A small compile-time investment that does some minimal optimization of unexercised code segments can yield a big win at run time if the program actually executes these code segments. Leveraging off of the loop grouping technology of Chapter 5 allows a significant number of loops that are unexercised during profiling to have their trip counts predicted.

One area of future work is to enhance the weaker confidence prediction functionality described in Section 6.2.3. This preliminary work is able to predict more loops than when only using the prediction technology from Section 6.2.2. However, there is room for improving the accuracy of these predictions. Enhancements include using other weak loop group members that are exercised to help predict unexercised weak loop group members, and allowing the number of conditions needed by a loop to continue to iterate to influence the loop-trip-count prediction.

Another important area of future work is knowing how to use the loop-trip-count predictions in unexercised code. As seen in Figure 6.5, it is possible for a very high percentage of loops to be unexercised during profiling. Because the number of unexercised loops can be large, understanding how to make a small compile-time investment in this code may be difficult. There needs to be some way to determine which set of unexercised loops would benefit the most from optimization. In addition, the actual optimizations that should be applied need to be determined.

CHAPTER 7

COMPILATION WITH VARYING PROGRAM BEHAVIOR

This chapter investigates the compilation of code when the program behavior observed during profiling does not match the behavior of the code during real program executions. The issue of the program's behavior differing for the profile inputs and the real inputs is always a concern when a compiler relies on profile information to guide compilation. This problem is difficult to address because it is difficult to anticipate portions of the program that are going to have varying behavior based on the inputs. The solution taken in this dissertation is to rely on static analysis and to use profile information when real trade-offs need to be made. If static analysis shows that an optimization can help certain executions and not hurt other executions, do the optimization regardless of the results of profiling. In addition, if static analysis shows that an optimization cannot help any executions and hurts other executions, do not apply the optimization. Profile information should be used to make decisions when static analysis shows that in order to help one execution, another execution must be hurt.

The proposed solution of this dissertation is not easy to obtain. In order for an optimizing compiler to better handle program behavior variations, each optimization needs to be analyzed individually. This analysis should determine what parameters are important to the optimization, and it should explain how these parameters can be estimated statically. In addition, the analysis needs to determine when there are real trade-offs and be able to reference the profile information to make the best decision. This dissertation applies these concepts to acyclic scheduling with the development of a new heuristic: speculative hedge. While the implementa-

tion is specific to acyclic scheduling, the concepts that are employed in the scheduling heuristic should be applied to other compiler optimizations.

The chapter is composed of the two sections. Section 7.1 discusses the problem of varying program behavior in more detail, and introduces a profile-dependent problem that occurs in the loop invariant code removal optimization. Section 7.2 describes the speculative hedge heuristic and provides results illustrating the effectiveness of this acyclic scheduling technique.

7.1 Motivation

The problem of varying program behavior is always a concern for a compiler that depends on the results of profiling to guide compilation. If the program behavior during profiling does not match the program behavior for real executions, the compiler may not be able to generate efficient code.

Fortunately, many programs exhibit behavior that does not vary much for different inputs. Fisher and Freudenberger showed that a set of UNIX utilities and benchmarks from SPEC CINT92 and SPEC CFP92 exhibited little difference in behavior for a set of applied inputs [25]. However, even in the benchmarks they evaluated, there are benchmarks that contain program segments that are very dependent on the input. The benchmark *sc* performs spreadsheet computations on a user-selected range of rows and columns. The behavior of the loops controlling the computations is fundamentally different if one input performs computations over one row and multiple columns, while another input performs computations over multiple rows and one column.

Two benchmarks, which are evaluated as a part of this dissertation, exhibit a great deal of profile dependence. Profile dependence of the benchmark *Pcode*, which is discussed in Sec-

tion 3.2, occurs because the inputs contain data with different characteristics. *Pcode* uses a hierarchical representation that includes a list representation for file I/O. The data read in for input 1 contains an average of 1 element per list, while input 2 contains an average of 116 elements per list. Therefore, the loops that iterate over the lists exhibit different characteristics for these inputs.

The other benchmark exhibiting profile dependence is the UNIX utility *grep*. This benchmark's execution is dominated by one loop, and a branch in that loop acts fundamentally different if the input is an ASCII file versus a binary file. This causes problems when a compiler tries to aggressively compile this loop assuming one input type (either ASCII or binary), and the other input gets applied the majority of the time by end users.

Figure 7.1 shows the important loop in *grep* that exhibits the varying behavior. For both inputs, the branches are resolved so that instructions 1, 2, 3, and 4 execute. The branch associated with instruction 4 causes the problems. The ASCII file never overflows the `linebuf` buffer, while the binary file, which does not contain many '\n' characters, ends up overflowing the buffer most of the time. Therefore, if an ASCII file is used to guide compilation, a superblock contains instructions 1, 2, 3, 4, and 5. However, if a binary file is used to guide compilation, a superblock only contains instructions 1, 2, 3, and 4.

The problem with loop invariant code removal occurs after the superblock loop gets unrolled. The superblock loop created using the binary file for profiling is unrolled eight times. If a binary file is applied as a real input, the program executes efficiently. However, if the ASCII file is applied as a real input, the program's performance suffers. The program ends up taking 73% longer to execute than it would have if the ASCII input had guided compilation. A similar

```
    for (;;) {
1:          if (p2 >= ebp)
                \* code originally found here in the source code has been removed *\
2:          if ((c = *p2++) == '\n')
                break;
3:          if (c)
4:          if (p1 < &linebuf[BUFSIZE–1])
5:              *p1++ = c;
    }
```

Figure 7.1 Important loop in *grep* that exposes a profile-dependence issue in the loop invariant code removal optimization.

phenomenon occurs if the ASCII file is used for profiling and the binary file is applied as a real input.

The problem occurs because loop invariant code removal forces eight add operations to be removed from the loop. These eight add operations are executed in the loop preheader before the loop is executed. When the opposite input case is applied, the superblock loop is exited, and the preheader needs to execute before the loop can be re-entered. In addition, the loop invariant code removal does not help the loop execute faster because the loop's execution is bounded by the retirement of the branches. The loop invariant code could have been left in the loop and not affected the execution time of the loop. Because the loop invariant code removal does not help performance, an unnecessary penalty is paid. This penalty does not hurt performance much if the branches in the loop are correctly predicted. However, if a branch is mispredicted, the penalty can be severe. In this example from *grep*, almost 50% of the extra cycles lost because of the mispredicted branch could have been avoided if loop invariant code removal had not been applied to the loop.

The *grep* example illustrates how a compiler heuristic can depend too much on profile data and end up unnessarily penalizing program executions that act differently than expected. The

assumption made by loop invariant code removal is that loops iterate many times and loop preheaders execute few times. While this is usually the case, there can be exceptions. Because of these exceptions, the loop invariant code removal heuristic should attempt to only apply its optimization when a "win" for the loop's execution can be determined. This is not an easy determination to make because of the information that is available to the compiler when loop invariant code removal is applied. However, the loop invariant code removal heuristic should do as good of a job as possible to avoid unnecessary application of its heuristic.

The following section describes the speculative hedge acyclic scheduling heuristic. It is an example of a compiler heuristic that tries to ensure that execution paths, which do not appear to be important based on the profile information, are not penalized unnecessarily.

7.2 Speculative Hedge

7.2.1 Introduction

Path-oriented scheduling methods such as trace scheduling [47], superblock [2], and hyperblock scheduling [3], extract instruction-level parallelism from control-intensive programs by using speculation. Profile information or frequency estimation guides aggressive speculation, so that important execution paths can have their run time minimized. However, with limited execution resources, situations arise where one path will execute faster, only if another path gets delayed.

Fisher proposed the use of *speculative yield* to determine the profitability of speculating an instruction [48]. The speculative yield is an expected value function that is defined between basic blocks. It is the probability that an operation scheduled in basic block$_i$ produces useful work (meaning that its original basic block$_j$ executes when basic block$_i$ executes). Its use with

dependence height has been shown to account for the needs of all paths during the scheduling process [5].

Speculative yield, coupled with dependence height, provides a good heuristic for path-oriented schedulers, but it does not address the problem of mismatches between compile-time prediction and run-time behavior. There is nothing inherent to speculative yield and dependence height that ensures that paths, which are shown by profile data to be unimportant, do not get delayed unnecessarily. This leads to execution-time slowdown when those paths are really executed at run time.

The *speculative hedge* heuristic attempts to ensure that no path gets delayed unnecessarily, even while performing aggressive speculation. Therefore, it *hedges* its reliance on the compile-time prediction data. It accomplishes this goal by accounting for different processor resources while scheduling, not just the common scheduling priority function of dependence height.

Speculative hedge only solves part of the profile independence problem. Speculative hedge does not address the problem of determining what paths should be scheduled together. A poor selection of paths to schedule together will limit the amount of ILP that can be generated for a program, and cause a profile-dependence problem [49]. Speculative hedge limits the damage associated with a poor selection by ensuring that paths are not delayed unnecessarily. This property is especially important for any compiler that uses static analysis instead of profile information to determine which paths should be scheduled together [33].

The usefulness of the speculative hedge heuristic is demonstrated in a superblock scheduler. The remainder of the section describes the heuristic, its implementation in the superblock scheduler, and the results obtained when the heuristic is applied to six SPEC CINT92 benchmarks. The remainder of this section is organized as follows. Section 7.2.2 provides background

on terms used in the section and related work. Section 7.2.3 provides the details of the speculative hedge heuristic. The performance results are reported in Section 7.2.4. A summary of speculative hedge is presented in Section 7.2.5.

7.2.2 Background

Dependence height is an important measure in determining an operation's scheduling priority. Many heuristics depend on it exclusively during scheduling. Speculative hedge uses dependence height as one component when determining scheduling priority. *Late times* are used by the speculative hedge heuristic to account for dependence height while scheduling.

A late time for an operation is the latest time that an operation can be scheduled without delaying an exit. An operation has a set of late times associated with it, one for each exit. Late times are calculated based on the latencies associated with the dependence graph used for scheduling. Figure 7.2 shows the late time values for a simple dependence graph. The late times are a 2-tuple in the figure. The first entry in the 2-tuple is the operation's late time for exit 0, and the second entry is the operation's late time for exit 1.

Late times are computed via a bottom-up traversal of the dependence graph. The calculation is done once for every exit. An exit operation has its late time defined as the dependence height of the exit, which would have been determined with a top-down traversal of the dependence graph, minus one. In the example, operation 2's late time for exit 0 is 1, and operation 5's late time for exit 1 is 3. All other operations have their late times computed based on the exit late times of operations located later in the dependence graph. Operation$_x$'s late time for exit$_i$ (LT_x) is defined in Equation 7.1.[1]

[1]In a practical implementation, if all dependence arcs go in the forward direction relative to the original program order, the equation can be applied once for each operation by visiting the operations in the reverse program order.

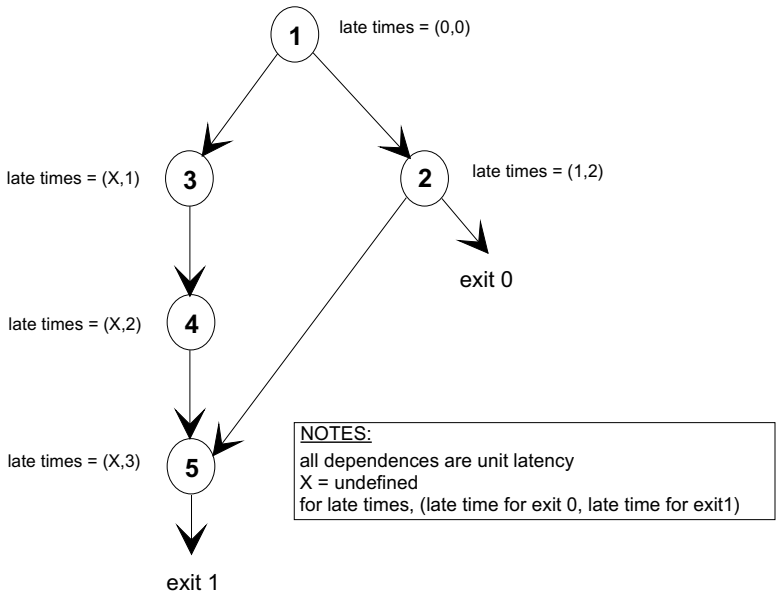

Figure 7.2 Dependence graph used for scheduling.

For all dependence arcs exiting operation$_x$ and entering any operation$_y$,

$$LT_x = \min_{for\_all\_ops_y} (LT_y - lat_{xy}) \tag{7.1}$$

where, LT_x = op$_x$ late time for exit$_i$

LT_y = op$_y$ late time for exit$_i$

lat_{xy} = latency of dependence arc between ops x and y

If an operation does not reach an exit through a dependence chain, the operation does not contain a *valid late time* for that exit. Therefore, its late time for the exit is undefined.

Speculative hedge utilizes late times in the conventional manner, but allows an exit operation's late time for that exit to be defined by the most constraining resource height, instead of always using dependence height. In Figure 7.2, if the processor being scheduled is single-issue, the exit associated with operation 5 cannot be issued as quickly as the dependence height dictates. Because issue width is the limiting resource, operation 5's late time for exit 1 would

actually be four (the issue width height minus one). Therefore, all the late times associated with exit 1 would be one greater than the values shown in the figure.

7.2.2.1 Dependence height and speculative yield (DHASY)

Fisher suggests that the multiplication of dependence height by speculative yield is a good candidate for the scheduling priority function [48]. The appeal is that dependence height is commonly used as the priority function, and speculative yield allows it to take into account the probability of taken branches.

Bringmann utilized this concept in a list scheduler for superblocks [5]. In his method, a static heuristic utilizes the exit probabilities[2] and an operation's late times to generate the priority for that operation. The operations are then scheduled greedily, so that the highest priority operation that is available gets scheduled. The priority function used in this scheduler is defined in Equation 7.2.

$$Priority_x = \sum_{ValLT_x} (Prob_i * (MaxLT + 1 - LT_x)) \tag{7.2}$$

where, $ValLT_x$ = for all valid late times$_i$ of op$_x$

$Prob_i$ = probability of taking exit$_i$

$MaxLT$ = maximum late time in dependence graph

LT_x = op$_x$ late time for exit$_i$

Figure 7.2 illustrates how the priority values are obtained. Assume exit 0 is taken 25% of the time and exit 1 is taken the other 75% of the time. The following priority values result:

$$\text{operation}_1 = 0.25 * (3 + 1 - 0) + 0.75 * (3 + 1 - 0) = 4$$

[2]Since a superblock has only one entrance point, exit probabilities are equivalent to Fisher's speculative yield values.

$\text{operation}_2 = 2.25$

$\text{operation}_3 = 2.25$

$\text{operation}_4 = 1.5$

$\text{operation}_5 = 0.75.$

However, using only dependence height to guide the scheduler can delay exits unnecessarily. This is shown in Figure 7.3. Assuming the latencies marked in the figure, the second branch, operation 7, can be retired in the third cycle when one only considers dependence height. If the profile information shows that exit 1 is always taken and dependence height guides the scheduling priority function, operations 1 and 2 would be scheduled in the first cycle. The earliest time that exit 1 can actually be retired is in the fourth cycle because there are eight operations that must be issued on this two-issue machine. Since the issue width of the processor is the limiting factor, exit 0 should be retired right away (regardless of the profile information). This allows exit 0 to be retired immediately, and it still allows exit 1 to be retired as soon as possible. This case illustrates the main principle behind speculative hedge: account for processor resources so exits are not delayed unnecessarily.

7.2.2.2 Critical path

Critical path is a profile-independent scheduling heuristic that has been used in many types of schedulers [50], [51], [52]. Its application to superblock scheduling consists of using the dependence height associated with the last exit in a superblock.

The DHASY priority function creates identical schedules to *critical path* if the profile information shows that the last exit has a probability of one and all other exits have a probability of zero. This scheduling heuristic suffers from overspeculation along the longest path of con-

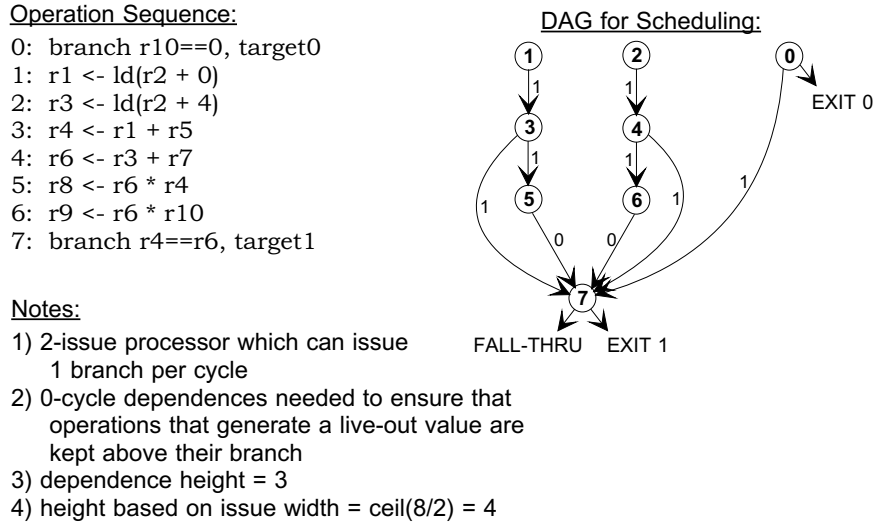

Figure 7.3 Example of dependence height not being a problem.

trol, and from the potential to delay earlier side exits unnecessarily. Speculative hedge can use the same profile-independent assumption that the last exit is always taken, and create better schedules through its use of resource information, along with the dependence information.

7.2.2.3 Successive retirement

Successive retirement is another profile-independent scheduling heuristic that attempts to retire each exit in order, as early as possible [53], [54]. This heuristic minimizes speculation, so that it only speculates when there are no nonspeculative instructions available.

This scheduling method works well for narrow-issue processors, where speculation is not important. However, it can potentially lose performance by not speculating enough for wide-issue processors. When a narrow-issue processor is scheduled for, successive retirement and speculative hedge create similar schedules. This results from speculative hedge's ability to understand when the processor's issue width is the limiting factor in the retirement of exits.

7.2.3 Speculative hedge heuristic

7.2.3.1 High-level overview

Speculative hedge takes both dependence height and resource constraints into account while scheduling. Many previous methods have considered only dependence height as the limiting factor. As shown previously, taking into account only dependence height can delay seemingly unimportant exits.

To account for the resources accurately, speculative hedge uses a dynamic-priority scheme. The dynamic-priority calculation is needed because exits can be constrained by different resources at various times during the scheduling process. The type of resource that constrains an exit's retirement has a direct effect on scheduling decisions. The compilation-time implications of this approach are addressed in Section 7.2.4.2.

In order to determine the best operation to schedule next, speculative hedge uses several priority values. The priority values used are:

Helped weight – The sum of the taken probabilities for all exits having a critical need met by an operation.

Helped count – The total number of exits having a critical need met by an operation.

Minimum late time difference – The minimum difference between the current cycle time being scheduled and any of an operation's late times.

Original program order – A unique number that is based on the original program location of an operation.

The first two priority values are utilized whenever the scheduling of an operation satisfies a *critical need* for an exit. A critical need can be dependence height, issue width, or any other restricted processor resource that must be dealt with in order for an exit to be retired by its *retirement goal*. An exit's retirement goal is the earliest time that an exit can be retired based on dependence height or any resource constraint. An operation satisfies a critical need by having the properties necessary to retire an exit quickly. One example of satisfying a critical need occurs when an operation has a late time equal to the current cycle being scheduled and an exit is dependence-height limited. Another example happens when an operation is an integer add operation and integer ALU issue width is the limiting factor for an exit. If an operation can satisfy a critical need, it *helps* an exit by allowing it to retire quickly.

Helped weight is the first criterion used to determine which operation should get scheduled. Helped weight's value is defined as the sum of all taken-exit profile weights that can be retired as soon as possible, if the current operation gets scheduled immediately. For example, if a superblock contains three exits (with each exit taken 25%, 40%, and 35% of the time respectively), an operation that helps the first two exits retire would have a helped weight of 0.65. An operation with the highest helped weight value gets scheduled next because it helps the most probable exits.

Of the four priority values, helped weight is the only one that depends on profile information. The other three priority values allow the speculative hedge heuristic to be less sensitive to profile variations. They allow fair decisions to be made for all exits, regardless of the compile-time predicted behavior.

The *helped count* priority value is used when operations share the same helped weight. Helped count is the number of exits having a critical need met by an operation. This criterion

gives zero-weight exits priority. If the weight of all the exits helped between two operations is equal, the operation helping the most exits should get scheduled first.

The helped weight and helped count priority values work together to prevent exits from being delayed unnecessarily. They accomplish this with the accurate accounting of resources. For example, if an important exit's retirement is issue-width limited, all operations located prior to the exit will meet the critical criteria for the exit. Therefore, operations helping an earlier, less important exit are critical for the important exit too. These operations would be chosen for scheduling because they will include both exits in their helped weight and helped count values. Even if the earlier exit was shown to never be taken based on the profile information, helped count would ensure the exit does not get delayed.

On the other hand, if dependence height is the limiting factor in an important exit's retirement, speculative hedge will work similar to the DHASY heuristic. Therefore, it will delay certain exits in favor of an exit that the profile information shows to be important. Speculative hedge does not keep exits from being delayed because of speculation. It only tries to minimize the number of exits and the penalty for exits that are delayed unnecessarily.

Another important aspect of speculative hedge is that it only looks at the exits that are helped by scheduling a particular operation. Speculative hedge's priority values do not directly deal with the condition where the scheduling of an operation might delay another exit. This is indirectly handled through the priority values for other operations. In order for the scheduling of operation$_x$ to delay an exit$_i$, there must exist another operation$_y$, whose scheduling would allow exit$_i$ to not be delayed. Since operation$_y$ helps exit$_i$, it gets exit$_i$'s contribution for helped weight and helped count. Therefore, the trade-off that needs to be made between the delaying

of an exit$_i$ and another exit is done via the comparison of operations x and y's helped weight and helped count values.

If the helped weight and helped count values are equal, the next criterion used is *minimum late time difference*. As defined earlier, each operation has a set of late times, one for each exit. An operation's late time difference for an exit is defined as the difference between the current cycle being scheduled and the operation's late time for that exit. An operation's minimum late time difference is the minimum of all the exits' late time differences.

This criterion is important because it helps anticipate which operations are about to become critical. Its biggest benefit comes when the helped weight and helped count values are both zero. This condition occurs after several operations have already been scheduled for the current cycle, all the scheduled operations have met all the exits' critical needs, and there is still at least one open slot in the current cycle.

The final criterion, original program order, is used when all the previous criteria have tied. It is used so that an operation located earlier in the program order gets scheduled first. This keeps unnecessary speculation from occurring and allows a deterministic schedule.

The priority values are used by speculative hedge to determine which operation should be scheduled next. The speculative hedge heuristic computes the priority values in the following way. First, the retirement goals and critical needs for every exit that has not been retired are determined. Then, all the late times for unscheduled operations are computed using the exit retirement goals as the defined late times for their corresponding exit operations. Finally, the priority values for all operations available for scheduling are computed, and the operation with the highest priority gets scheduled.

The remainder of this section describes the details needed to utilize the speculative hedge heuristic effectively. First, the determination of exit retirement goals and critical needs is presented. Second, the application of the priority value calculation is shown. Third, some potential problems with speculative hedge are discussed. Finally, a discussion of the issues specific to predication are presented.

7.2.3.2 Exit retirement goals and critical needs

The key components of speculative hedge are the estimation of the number of cycles that remain before an exit can be retired and the determination of which resources must be utilized in order to not delay the exit. This information is used directly by the dynamic priority function to determine whether an exit has a critical need, and whether the scheduling of an operation will enable that need to be met. The scheduling of an operation meeting a critical need either shortens the unscheduled dependence height or consumes a resource so that resource need in the future is alleviated. As with the priority calculation, this information is recomputed dynamically during the scheduling process.

In speculative hedge, the estimation of when an exit can be retired is optimistic. It is a lower bound on how quickly an exit can be retired [55]. An estimate may change during the course of scheduling because conflicting needs between different exits force some exits to delay. It is the dynamic priority function's responsibility to effectively deal with the trade offs that must be made during the scheduling process.

There are three main needs that are always considered, independent of the processor that is targeted. They are dependence height, the branch unit, and issue width. Other processor

resources that are restricted, such as memory operation issue width or special decoder requirements, are considered when they apply.

For each need, a cycle estimate is made that determines when an exit can be retired, based exclusively on that need. The maximum of all the cycle estimates for each need yields the retirement goal for an exit. Any need whose cycle estimate equals the exit retirement goal is considered critical.

The cycle estimate based on dependence height is a straightforward computation that determines the dependence height that must be honored before an exit can be retired. It must account for unscheduled operations and latencies for already scheduled operations that have not been fully satisfied.

The cycle estimate based on the branch unit is a function of the number of branches that can be issued in the same cycle, and of the retirement goals for previous exits sharing a common control path. Branches guarding a common control path cannot be reordered. Therefore, an exit with a branch must have a cycle estimate that is at least the same as the retirement goal for a preceding branch along a common control path. If an exit is a fall-thru path, it must have a retirement goal that is at least the same as any preceding exit's retirement goal.

If the number of preceding branches having the same retirement goal is equal to the number of branches that can be issued in any cycle, the current exit's cycle estimate is the preceding retirement exit goal plus one. This happens because there are not enough branch units to retire all of those exits simultaneously. In the one branch-per-cycle case, the cycle estimate based on the branch unit is always one more than any preceding exit retirement goal.

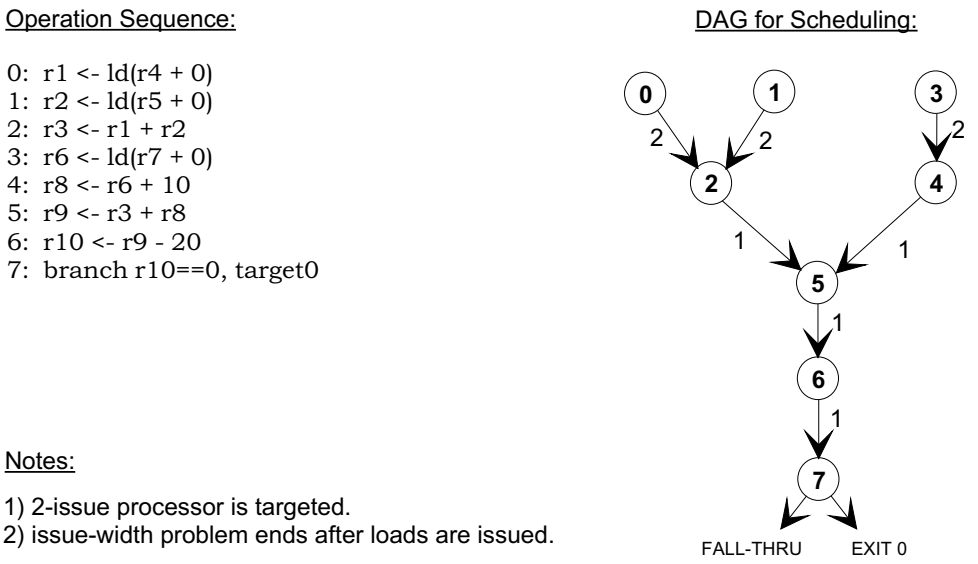

Figure 7.4 Example of an issue-width problem located at the beginning of a dependence chain.

Speculative hedge's cycle estimate based on issue width (ce_{iw}) could be the one presented in Equation 7.3.

$$ce_{iw} = \lceil num\_ops\_to\_sched\_before\_exit / issue\_width \rceil \tag{7.3}$$

However, this simple calculation does not detect enough issue-width problems. Many times, an exit may have an issue-width problem early in its dependence chain, but contain few operations later in its dependence chain. This is illustrated in Figure 7.4. In this example, a two-issue machine needs to execute three loads that, based on dependence height, must be executed in cycle zero in order to not delay the exit. Since it is impossible to issue all of them simultaneously, the exit is really issue-width limited at the beginning of its dependence chain.

To deal effectively with this, speculative hedge detects these cases, and keeps track of the late times that are included in any issue-width problem. This allows the speculative hedge heuristic to ensure that the issue-width problem is taken care of properly. For an operation

to help solve an issue-width problem, it must have a late time for the exit that is less than or equal to the late time where the issue-width problem ends.

The cycle estimates for other restricted resources, like a limited number of integer ALU operations that can be issued in any cycle, can be handled similarly to the general issue-width problem just discussed.

7.2.3.3 Priority calculation

Now that the exit retirement goals and critical needs have been determined, the dynamic priority calculation is performed. This dynamic scheme does require more computations than a static priority scheme, but is needed in order to account for all the changing resource requirements. The compilation-time implications of this method are discussed in the Section 7.2.4.2.

The priority calculation is performed for every operation that can be scheduled by evaluating each exit and determining whether the operation meets a critical need for the exit. If the scheduling of an operation meets a critical need, the operation helps the exit retire quickly. The high-level view of the priority calculation is shown in Figure 7.5.

If an exit_i has dependence height as its critical need, an operation_x meets that need if it has a late_time_i that is equal to the current cycle being scheduled. If operation_x meets the critical need, operation_x's helped weight and helped count get incremented for exit_i. A check is also made to determine if meeting of a critical need for the current exit_i will help future exits meet a critical need.

The check is needed to accurately account for branch-to-branch dependence height. A future exit_j is helped if the branch unit is a critical need for it, the current exit_i precedes it, it shares a common control path with the branch for exit_i, and delaying the branch for exit_i would cause

```
for (each unscheduled, available oper_x) {
    helped_weight_x = helped_count_x = 0
    for (each exit_i) helped_found_i = FALSE
    for (each exit_i where oper_x has a valid late time_i) {
        if (dependence_critical_for_i && (oper_x's late time_i == current_cycle)) {
            helped_found_i = TRUE
            account_for_future_branches
        }
        if (issue_width_critical_for_i && oper_x's late time is located before issue width problem ends) {
            helped_found_i = TRUE
            account_for_future_branches
        }
        /* the following is an example of what needs to be done for restricted resources */
        if (integer_alu_issue_width_critical_for_i && oper_x is an integer alu operation &&
           oper_x's late time is located before integer alu issue width problem ends) {
            helped_found_i = TRUE
            account_for_future_branches
        }
    }
    for (each exit_i)
        if (helped_found_i) {
            helped_weight_x += profile_weight_i
            helped_count_x++
        }
    determine_min_late_time_diff_for_oper_x
}
```

Figure 7.5 Algorithm for dynamic priority calculation.

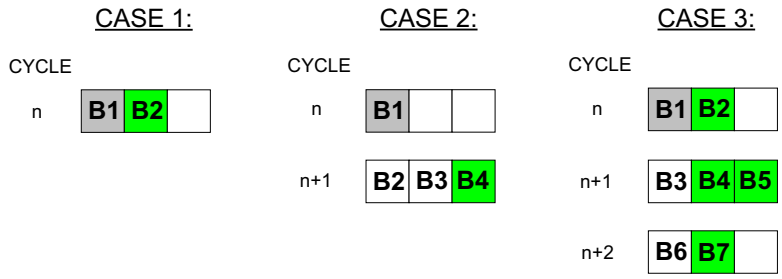

Figure 7.6 Cases for account_for_future_branches.

the future exit$_j$ to also be delayed. The determination of whether delaying exit$_i$ will also delay a future exit$_j$ is not trivial. Figure 7.6 helps to illustrate the different cases that are handled by speculative hedge.

For each case, branch B1 is associated with exit$_i$. It is the exit that has a critical need met by operation$_x$. A determination is made to see if any future exits$_j$ will also be helped by

operation$_x$. All branches shown are assumed to share a control path, and three branches can be issued in one cycle.

Case 1 is a basic example. In it, branch B2 is located after B1, and in the same cycle as B1. Therefore, if B1 gets delayed, B2 must be delayed too. B2's exit gets added to operation$_x$'s helped weight and helped count.

Case 2 is another basic example. Here, B1 is located by itself in a cycle, but the next cycle contains three branches. If B1 gets delayed, too many operations end up being located in the same cycle. So, B4 must be delayed also. B4's exit gets added to operation$_x$'s helped weight and helped count.

Case 3 is a combination of both cases 1 and 2. Here, the delay of B1 causes B2, B4, B5, and B7 to be delayed. Therefore, each one of them get their corresponding exit added into operation$_x$'s helped weight and helped count. In this cascading fashion, an operation$_x$ can help many exits just by meeting one exit's critical need.

The multiple branches-per-cycle problem is actually an important special case of a more general phenomenon. When predication is not used, branches have zero-cycle control dependences linking them. A similar case can also arise for other instructions. An example is a sequence of integer ALU operations that are linked by zero-cycle antidependences. If only one integer ALU operation can be issued in a single cycle, all the zero-cycle antidependences can be changed to one-cycle dependences in order to account for resource height. What makes branches the most interesting case is that, when predication is not used, they are always connected by these control dependences; no other instruction type has a zero-cycle dependence always linking them.

Operation$_x$ meets an exit's issue-width critical need if the exit has issue width as a critical need and if operation$_x$'s late time is less than the late time associated with the issue width

problem. The late time check is needed to ensure that the issue-width problem is really being addressed. As discussed in the previous section, an issue-width problem can be isolated to the beginning of an exit's dependence chain.

Finally, the checks needed for a restricted resource are also shown in Figure 7.5. They are identical to the issue width critical comparisons, with an extra check to ensure that operation$_x$ is the correct type of instruction. This same check can be replicated for any type of restricted resource that the scheduler should account for.

7.2.3.4 Potential problems

As with any scheduling heuristic, there are some potential areas of concern with speculative hedge. Scheduling is an NP-complete problem, and obtaining an optimal solution for all cases is not feasible.

One problem area of speculative hedge has to do with the trade-offs made while selecting an operation to schedule. An operation may be chosen because it helps an important exit$_i$, but scheduling it may delay a less important exit$_j$. The problem arises when a subsequent trade-off may mean that exit$_i$ gets delayed in order to help an even more important exit$_k$. Delaying exit$_j$, in hindsight, was unnecessary. Because the heuristic only looks at the situation for the current cycle, it may not always make the best decisions for the entire scheduling region as a whole.

This problem, while seen in practice, does not appear frequently enough to be a major issue. Also, it does not hurt the most frequently executed exits, and it causes only a short, unnecessary delay for the exits that it affects.

There is another problem related to scheduling trade-offs. This can happen when an important exit has dependence height as its critical need, and in the process of satisfying the dependence height need, it uses all the resources for a particular instruction type, like an integer ALU instruction. In addition, this situation lasts for several cycles. A less important exit can be retired immediately if it is allowed to issue an instruction whose type is being used up by the more important exit, but speculative hedge chooses to help the more important exit. When the problem is not solved for several cycles, the less important exit gets delayed several cycles.

This is an issue because a net loss may result. The cost of delaying the less important exit by several cycles may not be offset by allowing the more important exit to be retired one cycle early. Speculative hedge only looks at the costs involved for the current cycle; it does not look at the costs that are involved in the future.

A final problem deals with speculative hedge's desire to not consciously schedule for an exit's retirement until it is absolutely necessary to do so. Speculative hedge will delay an operation on a path shown by profile information to be taken 100% of the time for an operation on a 0% path, if it determines that scheduling for the 0% path will not delay the 100% path. This works fine as long as the speculative hedge heuristic makes no misjudgments concerning the 100% path. If a misjudgment is made, the 100% path may get delayed unnecessarily.

This problem was seen in several places during the evolution of the speculative hedge heuristic. As the heuristic was improved, the problem became less pronounced. The main mechanism in the heuristic that helped alleviate the problem was the *determine_min_late_time_diff_for_oper* function. This function identifies operations that are most likely to be needed next, and helps ward off the potential problem.

7.2.3.5 Dealing with predicated code

The work presented so far can be used for both predicated and unpredicated code. The heuristic has been designed so it can handle multiple paths of control in the same scheduling structure. However, there are some issues involving the accounting of resources that need to be dealt with when *fully-resolved predicates* are encountered [56].

A simple example illustrating the potential problem is shown in Figure 7.7. In this hyperblock [3], there are two threads of control. One thread takes more resources, and it is executed a few more times than the other. Because the threads are independent, there are no dependences between them, and any scheduling decisions made by the speculative hedge heuristic cannot account for the resources each thread uses individually. In the example, schedule 1 attempts to retire the bigger thread first, then retire the other thread. Schedule 2 attempts to retire them in reverse order. The better schedule is obtained in schedule 2, but the speculative hedge heuristic cannot obtain it given its current implementation.

The problem occurs because the bigger thread uses many resources — all the issue-width resources for three cycles — and when it gets scheduled for first, there are no more resources available for the smaller thread. This causes the smaller thread to be delayed for a large number of cycles. In this case, speculative hedge's problem is that it does not realize that a small penalty to the bigger thread can help the smaller thread greatly.

While this problem is possible, it should not be a common case for two reasons. First, two paths of control are probably not totally independent. If there are any dependences going between the control paths, speculative hedge can simultaneously account for some of the resources in both paths. Second, one control path must need all the available resources for many cycles.

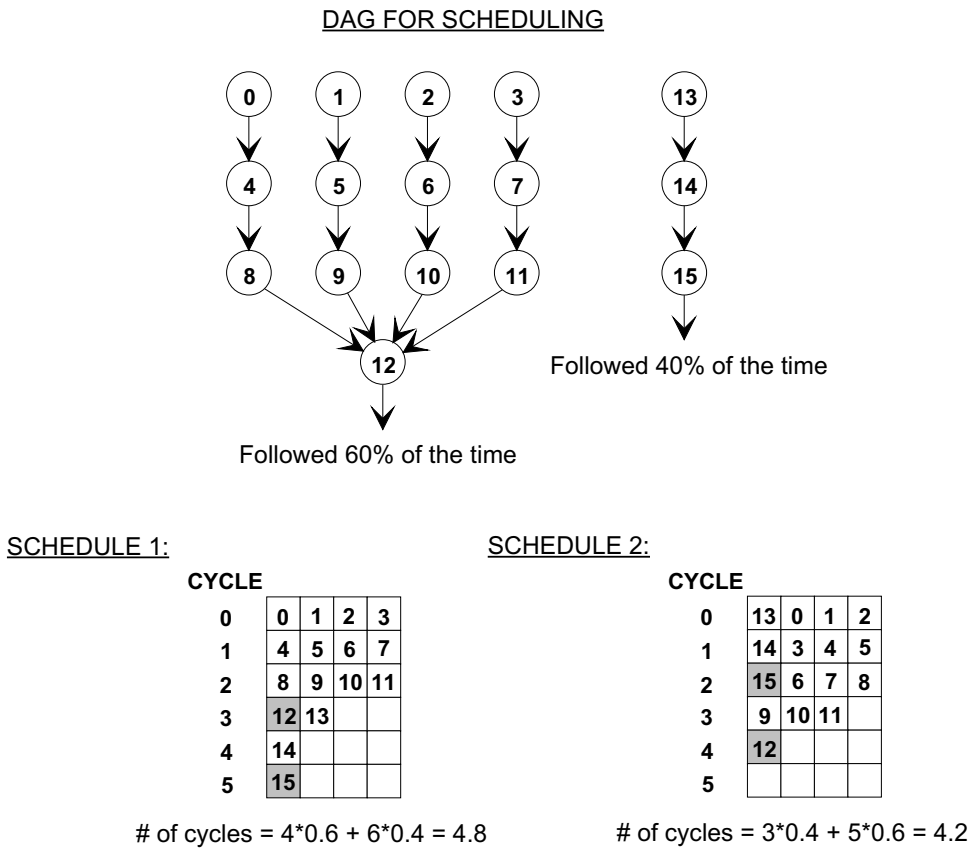

Figure 7.7 Problem of resource accounting in a hyperblock.

Speculative hedge attempts to schedule for resources as late as possible, so less important paths have a chance to use resources early.

However, scheduling heuristics need to ensure that worst-case scenarios are avoided. A means for speculative hedge to overcome this potential problem is to mark, prior to scheduling, the expected exit retirement goal for each exit. Then after scheduling, the actual cycle in which the exits are retired can be compared against the cycle based on the exit retirement goal. If the actual cycle in which an exit is scheduled to retire is three cycles or 10% later than expected, whichever is greater, corrective action may be necessary. Corrective action means that a postpass of the resulting schedule is performed. If a case similar to the one found in Figure 7.7 is detected, artificial branch-to-branch dependences can be added to ensure that one

exit is scheduled before another. Then the hyperblock is rescheduled. The artificial dependences are only added to help the scheduling heuristic, and as long as they are added carefully, they should not be a detriment to obtaining an efficient schedule.

7.2.4 Experimental analysis

In this section, the effectiveness of the speculative hedge heuristic in minimizing the unnecessary delaying of exits is analyzed for a set of SPEC CINT92 benchmarks.[3] First, speculative hedge's performance is compared against other scheduling heuristics on real input cases for a variety of machine configurations and compile-time assumptions. Then, an in-depth analysis is done to show exactly how well the speculative hedge heuristic controls unnecessary speculation for an exit that does not appear to be important based on the compile-time predicted data. Finally, the compilation-time implications of speculative hedge's dynamic priority scheme are addressed.

7.2.4.1 Methodology

The speculative hedge, DHASY, and successive retirement heuristics have been implemented in the IMPACT compiler [58]. This compiler inlined, coalesced into superblocks, and fully ILP optimized the six SPEC CINT92 benchmarks used in this testing. The benchmarks used are *espresso*, *li*, *eqntott*, *compress*, *sc*, and *cc1*.

To show the effectiveness of the speculative hedge heuristic for various issue width and functional unit constraints, four machine models are utilized:

[3] These benchmarks are different from the ones used when presenting the results for the rest of this dissertation. These benchmarks are the ones presented in the MICRO paper that first described the speculative hedge acyclic scheduling heuristic [57].

Table 7.1 Instruction latencies.

Function	Latency	Function	Latency
Int ALU	1	FP ALU	2
memory load	2	FP multiply	2
memory store	1	FP divide(SGL)	8
branch	1 / 1 slot	FP divide(DBL)	15

4_MIX – A four-issue processor that contains two memory ports, two integer ALUs, one floating point ALU, and one branch unit. This machine model is similar to many of today's processors.

4_1BR – A fully-uniform four-issue processor that contains only one branch unit. This machine model allows more scheduling freedom than 4_MIX and shows how the scheduling heuristics perform when the functional units are not constraining resources.

4_2BR – A fully-uniform four-issue processor that contains two branch units. This machine model forces the condition where branch resource height is difficult to account for.

SINGLE – A single-issue processor. This machine model forces issue width to be a major issue and speculation to be minimized in order to obtain good performance.

Each machine model's latencies match those of the HP PA-RISC PA7100 processor, and the latencies are shown in Table 7.1. An enhanced version of the HP PA-RISC processor instruction set is used with a nontrapping version of instructions added, so speculative instructions cannot cause program termination. This allows the general speculation model to be utilized by the scheduler [58]. This model gives the scheduler the freedom and choices during scheduling to help differentiate the performance of the heuristics.

The analysis focuses on prepass scheduling and on the effect of compile-time predicted behavior on the quality of the schedules produced. The effects of register allocation, cache misses, branch mispredictions, etc. are factored out, so that a fair comparison can be made between the different scheduling heuristics. This means that the execution time of programs is determined statically, by knowing the schedule time for a particular exit and the number of times that the profile information shows that the exit is taken.

7.2.4.2 Results

This section compares the schedules obtained for the different benchmarks. The first set of results is used to analyze the schedules obtained by the speculative hedge, DHASY, and successive retirement heuristics for the four different machine models and different compile-time profile assumptions. The different schedules are compared assuming that the actual program input matches the exact input used to generate one of the compile-time profile assumptions. The second set of results is used to analyze the effectiveness of the speculative hedge and DHASY heuristics to schedule the first exit when they both assume that the last exit is always taken. Finally, the time needed to schedule with the speculative hedge heuristic is compared against the time needed to schedule with DHASY.

Evaluation for real execution of programs

In this test, the effectiveness of the speculative hedge heuristic in controlling unnecessary speculation is analyzed. This is done by scheduling each benchmark on four different machine models with the different scheduling heuristics. In addition, the two profile-dependent heuristics, speculative hedge (SH) and DHASY, are given three different compile-time profile

assumptions. The different assumptions will be used to show that SH is less sensitive to profile variations than DHASY.

The three different profile-time assumptions are *REAL* (the profile data match the program behavior for the real data that are input), *ALL1* (the profile data show that all exits are equally likely), and *LAST1* (the profile data show that only the last exit in the superblock is ever taken). Note that the critical path scheduling heuristic is the same as scheduling DHASY with the LAST1 profile assumption.

The results for the four machine models after scheduling with the different scheduling heuristics are shown in Table 7.2. The cycle count for SH-REAL assumes that the same profile input used to generate the SH-REAL schedule is used in the actual run. The *% diff* columns represent the difference between the column's scheduling heuristic cycle count and the SH-REAL's cycle count for the same benchmark and input data. A positive value means the column's heuristic required more cycles to execute.

SH is less reliant on the compile-time profile assumption than DHASY. The largest difference between SH-REAL and SH-ALL1/SH-LAST1 for any machine model and benchmark is 2%. This is contrasted with DHASY-REAL and DHASY-LAST1, which is critical path, that vary by over 5% for at least one benchmark in every machine model, and by over 10% for at least one benchmark for the 4_2BR and SINGLE machine models.

In addition, SH-ALL1 and SH-LAST1, which were scheduled with compile-time profile assumptions that differed from the run-time execution, outperformed DHASY-REAL for most of the test cases. SH-ALL1 and SH-LAST1 executed more cycles than DHASY-REAL for only one benchmark in each of the 4_MIX, 4_1BR, and 4_2BR machine models. SH-ALL1 and SH-LAST1 performed better because they were able to account for resources while scheduling

Table 7.2 Results when input data match input used for REAL profiling.

Machine Model	Benchmark	SH REAL # of cycles	SH ALL1 % diff	SH LAST1 % diff	DHASY REAL % diff	DHASY ALL1 % diff	Crit. Path[a] % diff	Succ. Ret. % diff
4_MIX	espresso	152405400	+0.5	+0.9	+0.2	+0.6	+2.1	+0.7
	li	14486519	0.0	0.0	0.0	+0.7	+0.3	+0.4
	eqntott	423624658	0.0	0.0	0.0	0.0	0.0	0.0
	compress	25767022	+0.1	+0.1	+0.6	+3.1	+8.2	+2.6
	sc	38204165	0.0	0.0	+0.5	+0.9	+0.3	+3.7
	cc1	49349343	+0.2	+0.1	+0.3	+0.7	+0.2	+1.1
4_1BR	espresso	146674421	+0.1	0.0	+1.4	+1.2	+1.0	+1.6
	li	13702023	0.0	0.0	+0.8	+1.3	+0.2	+1.1
	eqntott	401064104	0.0	0.0	0.0	0.0	0.0	0.0
	compress	24174638	+0.5	+0.5	+0.4	+3.4	+6.3	+2.9
	sc	36938586	0.0	0.0	+0.1	+0.3	0.0	+2.5
	cc1	47497736	0.0	0.0	+0.1	+0.2	+0.1	+0.7
4_2BR[b]	espresso	129275735	+0.1	0.0	+1.0	+1.9	+7.2	+2.3
	li	12389729	+0.7	+0.7	0.0	+2.2	+1.3	+2.0
	eqntott	313622471	+0.4	+0.4	+1.0	+4.0	+6.4	+0.9
	compress	22011735	+0.3	+0.3	+0.6	+7.1	+11.4	+2.7
	sc	30904853	+1.2	+1.1	+1.4	+2.0	+5.8	+4.6
	cc1	41450987	0.0	0.0	+0.4	+0.6	+3.7	+1.3
SINGLE	espresso	330192736	+0.2	+0.2	+1.6	+3.1	+11.2	+0.2
	li	27242181	+0.6	+0.6	+0.3	+1.9	+4.4	+1.9
	eqntott	826767804	0.0	0.0	0.0	+0.1	+0.1	+0.1
	compress	56980582	+2.0	+2.0	+0.7	+6.3	+15.6	−0.1
	sc	84079061	+0.5	+0.4	−0.2	+0.8	+1.0	+0.7
	cc1	99468982	+0.3	+0.2	+0.1	+1.4	+3.1	+0.4

[a] Critical path is the same as DHASY-LAST1.
[b] All branch-to-branch dependences for DHASY are assumed to be 0.

and minimize the effect of exits that were delayed unnecessarily. Even though SH-ALL1 and SH-LAST1 may not have helped some of the execution paths that were important at run time (because they did not have that information), they were able to minimize unnecessary exit delays. On the other hand, DHASY-REAL did delay some less important exits unnecessarily, and this resulted in its schedule taking a few more cycles to execute than the schedules for SH-ALL1 and SH-LAST1.

To understand how speculative hedge performs for special cases, the SINGLE and 4_2BR machine models must be examined. The SINGLE machine model case shows how well spec-

ulative hedge performs when issue width is a severe problem. Speculative hedge generates schedules that are as good as or better than successive retirement. This highlights speculative hedge's ability to control speculation, so exits are not delayed unnecessarily.

In the 4_2BR machine model case, speculative hedge's method of accounting for the branch resource height allows it to be insensitive to the accounting problems for branch-to-branch dependences that effect DHASY. The zero-cycle branch-to-branch dependence assumption made by DHASY leads to its profile sensitivity problems. For this machine model, DHASY-LAST1 (critical path) varies an average of over 5% when compared to DHASY-REAL, while SH-LAST1 only varies an average of 0.4% when compared to SH-REAL.

Evaluation of the unnecessary delaying of an exit

The results shown so far have been for real execution cases. These results can be affected by a minority of the superblocks present in the benchmarks. This makes it difficult to show how large a problem the dependence on profile information can be. To illustrate the potential problem, speculative hedge and DHASY are scheduled assuming that the last exit is always taken. Then, a comparison is made between their resulting schedules showing how well they handle the first exit for all the superblocks in the six benchmarks. The first exit was chosen because it has the best chance of being delayed unnecessarily by over speculation.

The results are shown in Figure 7.8, and they were obtained for the 4_MIX machine model. The cycle difference subtracts the scheduled cycle for SH-LAST1's first exit to the cycle obtained for DHASY-LAST1's first exit. The zero cycle's column represents instances where SH-LAST1 and DHASY-LAST1 scheduled the first exit in the same cycle, and the cycle was later than what could have been obtained if the first exit was scheduled to retire as quickly as possible.

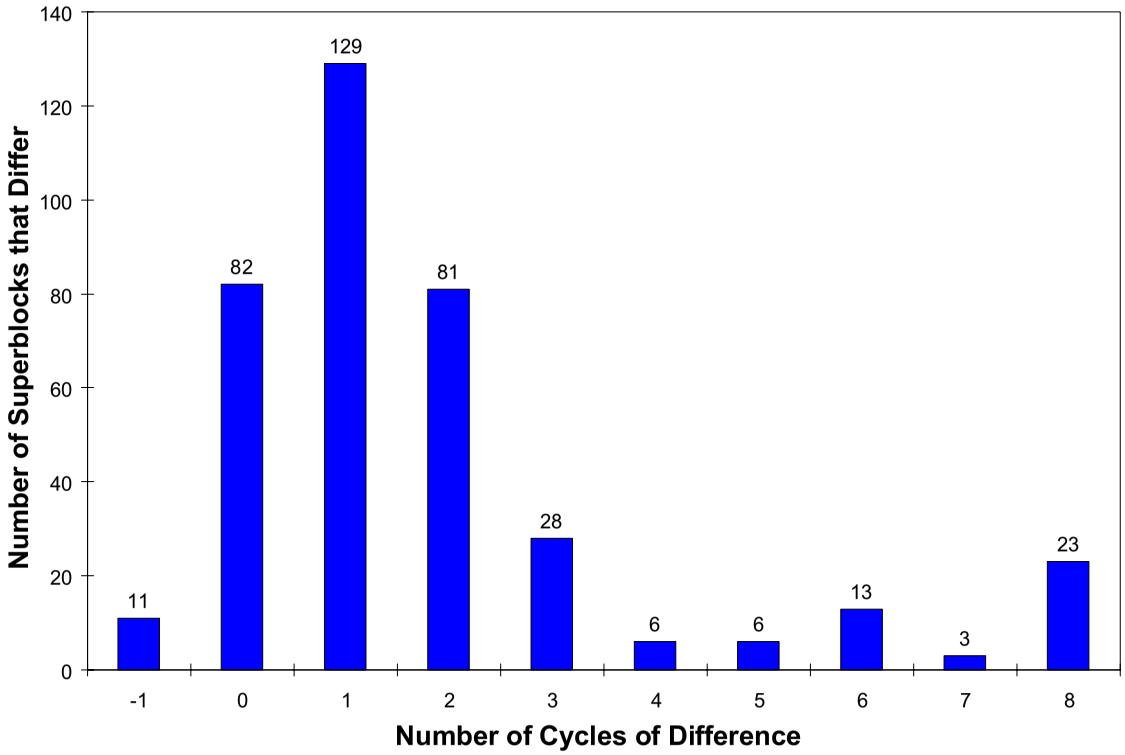

Figure 7.8 Cycle differences between SH-LAST1 and DHASY-LAST1 for the first exit.

As shown in the figure, DHASY does a worse job in retiring the first exit of the superblocks for all the benchmarks.

A comparison of SH's and DHASY's first exit schedule times to the earliest time that the first exit could be retired, which would be obtained if the profile information showed that it was the only exit ever taken, provides more insight into this issue. It confirms that not only does DHASY delay more exits unnecessarily, it also delays them longer than speculative hedge. An examination of the detailed data shows that, compared to the case where the first exit was retired as soon as possible, speculative hedge delayed 125 first exits for an average of 1.3 cycles, while DHASY delayed 379 first exits for an average of 2.4 cycles.

Evaluation of compilation time

The compilation time for speculative hedge must be addressed because of its use of a dynamic priority calculation. The priority calculation is performed on every operation that can be scheduled, which means there are no resource conflicts and all the operation's incoming dependences have been satisfied, and the calculation gets recomputed before any operation is scheduled. In addition, the priority calculation's run time is proportional to the number of exits that have yet to be retired. This leads to a worst-case run time for speculative hedge of $O(n^2m)$, where n = number of operations and m = number of exits. Measurements, during prepass scheduling of the SPEC CINT92 benchmarks, showed that speculative hedge had to compute priorities for an average of 2.8 operations before an operation got scheduled. This leads to a run-time of $2.8nm$ in practice. The worst-case run time for DHASY's static priority calculation is $O(nm)$.

The speculative hedge heuristic was implemented in the IMPACT compiler. Timing measurements were made during prepass scheduling. Besides the calculation of operation priorities and the actual scheduling, the only other computationally-intensive functionality performed was the data-flow analysis required to compute the live-out variables. IMPACT uses a standard, iterative data-flow algorithm for the analysis that has a worst-case run time of $O(b^2)$, where b = number of basic blocks [24]. For the benchmarks used in this section, prepass scheduling took 26% to 49% longer for speculative hedge than it did for DHASY.

7.2.5 Summary

This section described a new path-oriented, scheduling heuristic, speculative hedge, that attempts to minimize speculation so paths of execution do not get delayed unnecessarily. This

ensures that speculative hedge is much less sensitive to variations between compile-time predicted behavior and actual run-time behavior. This is done by accounting for dependence height and processor resources while scheduling. Previous scheduling heuristics that only accounted for dependence height while scheduling would delay exits unnecessarily and perform poorly at run time when the program executed differently than expected. The speculative hedge heuristic was implemented in a superblock scheduler to illustrate its effectiveness.

The section investigated the performance of speculative hedge for six programs from the SPEC CINT92 benchmark suite for four different machine models. Speculative hedge was scheduled with three different sets of compile-time predicted behaviors, and the largest variation for any combination of machine model and compile-time predicted behavior was 2%. In contrast, DHASY, using the same sets of compile-time predicted behavior, had variations of over 5% for at least one benchmark in every machine model, with three cases having variations of over 10%. In addition, an analysis of how the first exits of superblocks can be delayed for different profile inputs showed that DHASY delayed some superblock first exits by up to eight cycles more than speculative hedge.

The same concepts used in the design of the speculative hedge acyclic scheduling heuristic — statically accounting for the processor's resources and minimizing reliance on profile information — should be applied to other compiler heuristics. Through the use of static analysis, the resultant code produced by the compiler can perform better if the program behavior assumed at compile time does not match the program behavior at run time.

CHAPTER 8

CONCLUSION

8.1 Summary

Program profiling has been shown to be effective for many aggressive ILP optimizations. Profiling the program with test inputs allows the compiler to gain an understanding of how the program may behave during real executions. However, several issues associated with profiling must be addressed before aggressive ILP optimizations can be included in a production compiler. These issues include dealing with the case when the programmer chooses not to profile the program, overcoming the issue of the profile inputs not exercising all the functionality that is used in a program, and enabling good performance when the branch behavior observed during profiling differs from the behavior seen for real executions. This dissertation provides a groundwork for an ILP compiler to effectively deal with each of these issues.

In order to deal with the problem of profiling not being performed, this dissertation investigated static analysis techniques that inferred the information normally obtained from profiling. The major portion of this work involved static branch prediction. This dissertation improved the state of the art in static branch prediction and provided insights into the mechanisms that allow these predictions to work effectively. In addition, an analysis of the issues involved with static loop-trip-count prediction and static frequency generation was performed.

The problem of not exercising all important program functionality during profiling can be an issue when the profile inputs are chosen poorly or the time needed to profile certain

functionality is unacceptably large. To handle the issue, this dissertation introduced a new analysis technique, loop grouping, that allowed counted and linked list loops that iterate with the same loop control to be identified. The results showed that, in most cases, a majority of the loops important to program performance can be grouped with other loops. There are many potential useful applications for this technology beyond the use described in this dissertation.

In this dissertation, loop grouping was shown to help the problem of static loop-trip-count prediction in unexercised code. Loop grouping was used to statically predict the trip counts for loops that were unexercised during profiling. This allowed the compiler to extract information about the input data observed during program profiling, and apply it to the remainder of the program. Lack of knowledge about the input data was the reason that the general solution to the static loop-trip-count prediction problem was so difficult to obtain. For most of the benchmarks with at least one input not exercising functionality used by another input, a significant number of loop trip counts can be correctly predicted with this method.

The problem of the branch behavior differing between runs was addressed by using static analysis to limit the compiler's dependence on profile data. The idea was that a compiler heuristic minimized its dependence on profile information and ensured that the program's performance was not degraded unnecessarily when the program behaved differently than expected. This concept was applied to a new acyclic scheduling heuristic developed in this dissertation, speculative hedge. Speculative hedge was shown to create schedules that perform as well, or better, than schedules produced by prior techniques when the program behaved as expected. When the program behavior varied, speculative hedge's schedules were significantly better because the delays were minimized for paths shown by profiling to be unimportant.

This dissertation provided the groundwork necessary for a production ILP compiler to deal with the issues associated with profiling. The static analysis techniques developed facilitate the successful integration of aggressive ILP techniques into a production compiler. In order for future processors to reach their full potential, compilers will need to utilize many aggressive ILP optimizations.

8.2 Future Work

This dissertation investigated the fundamental issues that a compiler must handle when using profile information. Static analysis techniques were proposed that facilitated the successful integration of aggressive ILP optimizations into a production compiler. However, there is more work that is necessary before production compilers can successfully utilize these techniques.

Analysis techniques for loops that are not counted loops or linked list loops should be investigated. Loop grouping successfully dealt with counted loops and linked list loops, but other loop types can account for a significant amount of processing time. The design and implementation of static loop-trip-count heuristics to handle argument, file, and string processing loops would handle many of these other loop types.

Improvements to the static loop-trip-count prediction of weak correlation loop group members should be investigated. Section 6.2.3 examined a means of generating weaker confidence predictions, but more work is necessary in this area. For example, the technique can be selectively applied based on the behavior of other weak correlation loop group members that have been exercised. This may help limit the number of mispredicts that can occur with these "weaker" predictions.

This dissertation made acyclic scheduling less dependent on profile information, but other compiler heuristics should also be modified to ensure they make good, profile-independent decisions. While the actual techniques used in the speculative hedge acyclic scheduling heuristic may not be applicable to other heuristics, the concept of relying on both static analysis and profile data is applicable. One compiler optimization that can exhibit poor profile dependence behavior is loop invariant code removal, and Section 7.1 discussed the specific problems associated with this optimization.

The most obvious future work involves the integration of the static analysis techniques of this dissertation into the heuristics of a real compiler. The techniques have been shown to be effective in predicting program behavior, but how the compiler actually utilizes this information in order to obtain better performance across a wide range of inputs is still an open question.

The final area of future work involves the other application areas of loop grouping described in Section 5.3.1. Loop grouping is a new analysis technique developed in this dissertation that has potential uses beyond static loop-trip-count prediction of unexercised loops. These other applications include static prediction of non-loop branches unexercised during program profiling, allowing the compiler to override profile information when a loop group exhibits inconsistent loop-trip-count behavior, providing better feedback to users when they are developing inputs for profiling, and improving code maintenance by providing a list of loops that should be examined after fixing a bug in one loop.

REFERENCES

[1] P. P. Chang, S. A. Mahlke, and W. W. Hwu, "Using profile information to assist classic code optimizations," *Software Practice and Experience*, vol. 21, pp. 1301–1321, December 1991.

[2] 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.

[3] 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*, December 1992, pp. 45–54.

[4] 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.

[5] R. A. Bringmann, "Compiler-controlled speculation," Ph.D. dissertation, Department of Computer Science, University of Illinois, Urbana, IL, 1995.

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

[7] D. M. Gallagher, "Memory disambiguation to facilitate instruction-level parallelism compilation," Ph.D. dissertation, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1995.

[8] B.-C. Cheng and W. W. Hwu, "Multirelation alias graph: A new alias representation to analyze pointers in C programs effectively and efficiently," Tech. Rep. IMPACT-98-02, IMPACT, University of Illinois, Urbana, IL, February 1998.

[9] K. Subramanian, "Loop transformations for parallel compilers," M.S. thesis, Department of Computer Science, University of Illinois, Urbana, IL, 1993.

[10] S. Anik, "Architectural and software support for executing numerical applications on high performance computers," Ph.D. dissertation, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1993.

[11] Y. Yamada, "Data relocation and prefetching for programs with large data sets," Ph.D. dissertation, Department of Computer Science, University of Illinois, Urbana, IL, 1995.

[12] B.-C. Cheng, "Pinline: A profile-driven automatic inlines for the impact compiler," M.S. thesis, Department of Computer Science, University of Illinois, Urbana, IL, 1996.

[13] S. A. Mahlke, "Design and implementation of a portable global code optimizer," M.S. thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1991.

[14] S. A. Mahlke, "Exploiting instruction level parallelism in the presence of conditional branches," Ph.D. dissertation, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1995.

[15] P. P. Chang, D. M. Lavery, S. A. Mahlke, W. Y. Chen, and W. W. Hwu, "The importance of prepass code scheduling for superscalar and superpipelined processors," *IEEE Transactions on Computers*, vol. 44, pp. 353–370, March 1995.

[16] N. J. Warter, "Modulo scheduling with isomorphic control transformations," Ph.D. dissertation, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1993.

[17] D. M. Lavery, "Modulo scheduling for control-intensive general-purpose programs," Ph.D. dissertation, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1997.

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

[19] D. M. Gallagher, W. Y. Chen, S. A. Mahlke, J. C. Gyllenhaal, and W. W. Hwu, "Dynamic memory disambiguation using the memory conflict buffer," in *Proceedings of 6th International Conference on Architectual Support for Programming Languages and Operating Systems*, October 1994, pp. 183–193.

[20] R. E. Hank, "Machine independent register allocation for the IMPACT-I C compiler," M.S. thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1995.

[21] J. C. Gyllenhaal, "A machine description language for compilation," M.S. thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1994.

[22] R. G. Ouellette, "Compiler support for SPARC architecture processors," M.S. thesis, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1994.

[23] V. Kathail, M. S. Schlansker, and B. R. Rau, "HPL PlayDoh architecture specification: Version 1.0," Tech. Rep. HPL-93-80, Hewlett-Packard Laboratories, Palo Alto, CA, February 1994.

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

[25] J. A. Fisher and S. M. Freudenberger, "Predicting conditional branch directions from previous runs of a program," in *Proceedings of 5th International Conference on Architectual Support for Programming Languages and Operating Systems*, October 1992, pp. 85–95.

[26] T. Ball and J. R. Larus, "Branch prediction for free," in *Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, June 1993, pp. 300–313.

[27] J. E. Smith, "A study of branch prediction strategies," in *Proceedings of the 8th International Symposium on Computer Architecture*, May 1981, pp. 135–148.

[28] J. Lee and A. J. Smith, "Branch prediction strategies and branch target buffer design," *IEEE Computer*, pp. 6–22, January 1984.

[29] S. Bandyopadhyay, V. S. Begwani, and R. B. Murray, "Compiling for the crisp microprocessor," in *Proceedings of the IEEE Spring Compcon 87*, February 1987, pp. 96–100.

[30] Y. Wu and J. R. Larus, "Static branch prediction and program profile analysis," in *Proceedings of the 27th Annual International Symposium on Microarchitecture*, December 1994, pp. 1–11.

[31] G. Shafer, *A Mathematical Theory of Evidence*. Princeton, NJ: Princeton University Press, 1976.

[32] B. Calder, D. Grunwald, D. Lindsay, J. Martin, M. Mozer, and B. Zorn, "Corpus-based static branch prediction," in *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*, June 1995, pp. 79–92.

[33] R. E. Hank, S. A. Mahlke, R. A. Bringmann, J. C. Gyllenhaal, and W. W. Hwu, "Superblock formation using static program analysis," in *Proceedings of the 26th Annual International Symposium on Microarchitecture*, December 1993, pp. 247–255.

[34] T. A. Wagner, V. Maverick, S. L. Graham, and M. A. Harrison, "Accurate static estimators for program optimization," in *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation*, June 1994, pp. 85–96.

[35] J. R. C. Patterson, "Accurate static branch prediction by value range propagation," in *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation*, June 1995, pp. 67–78.

[36] D. E. Knuth, *The Art of Computer Programming*, vol. 1. Reading, MA: Addison-Wesely, 1973.

[37] N. Deo, *Graph Theory with Applications to Engineering and Computer Science*. Englewood Cliffs, NJ: Prentice-Hall, 1974.

[38] C. V. Ramamoorthy, "Discrete Markov analysis of computer programs," in *Proceedings of the 20th ACM National Conference*, August 1965, pp. 386–392.

[39] J. Kral, "One way of estimating frequencies of jumps in a program," *Communications of the ACM*, vol. 11, pp. 475–480, July 1968.

[40] B. Beizer, "Analytical techniques for the statistical evaluation of program running time," *AFIPS Conference*, vol. 37, pp. 519–524, 1970.

[41] K. S. Trivedi, *Probability and Statistics with Reliability, Queuing, and Computer Science Applications*. Englewood Cliffs, NJ: Prentice-Hall, 1982.

[42] D. W. Wall, "Predicting program behavior using real and estimated profiles," in *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*, June 1991, pp. 59–70.

[43] U. Banerjee, R. Eigenmann, A. Nicolau, and D. A. Padua, "Automatic program parallelization," *Proceedings of the IEEE*, vol. 81, pp. 211–243, February 1993.

[44] D. F. Bacon, S. L. Graham, and O. J. Sharp, "Compiler transformations for high-performance computing," *ACM Computing Surveys*, vol. 26, pp. 345–420, December 1994.

[45] T. Cormen, C. Leiserson, and R. Rivest, *Introduction to Algorithms*. Cambridge, MA: MIT Press, 1990.

[46] R. E. Hank, "Region based compilation," Ph.D. dissertation, Department of Electrical and Computer Engineering, University of Illinois, Urbana, IL, 1996.

[47] J. A. Fisher, "Trace scheduling: A technique for global microcode compaction," *IEEE Transactions on Computers*, vol. C-30, pp. 478–490, July 1981.

[48] J. A. Fisher, "Global code generation for instruction-level parallelism: Trace scheduling-2," Tech. Rep. HPL-93-43, Hewlett-Packard Laboratory, Palo Alto, CA, June 1993.

[49] T. M. Conte, B. A. Patel, K. N. Menezes, and J. S. Cox, "Hardware-based profiling: An effective technique for profile-driven optimization," *International Journal of Parallel Programming*, vol. 24, pp. 187–206, April 1996.

[50] C. V. Ramamoorthy and M. Tsuchiya, "A high level language for horizontal microprogramming," *IEEE Transactions on Computers*, vol. C-23, pp. 791–802, August 1974.

[51] S. Davidson, D. Landskov, B. D. Shriver, and P. W. Mallett, "Some experiments in local microcode compaction for horizontal machines," *IEEE Transactions on Computers*, vol. C-30, pp. 460–477, July 1981.

[52] W. H. Kohler, "A preliminary evaluation of the critical path method for scheduling tasks on multiprocessor systems," *IEEE Transactions on Computers*, vol. C-24, pp. 1235–1238, December 1975.

[53] C. Chekuri, R. Motwani, R. Johnson, B. Natarajan, B. R. Rau, and M. Schlansker, "Profile-driven instruction level parallel scheduling with applications to super blocks," in *Proceedings of the 29th Annual International Symposium on Microarchitecture*, December 1996, pp. 58–67.

[54] M. D. Smith, "Architectural support for compile-time speculation," in *The Interaction of Compilation Technology and Computer Architecture*, D. L. Lilja and P. L. Bird, Eds., Boston, MA: Kluwer Academic, 1994, pp. 13–49.

[55] B. R. Rau, "Iterative modulo scheduling: An algorithm for software pipelining loops," in *Proceedings of the 27th International Symposium on Microarchitecture*, December 1994, pp. 63–74.

[56] M. Schlansker and V. Kathail, "Critical path reduction for scalar programs," in *Proceedings of the 28th International Symposium on Microarchitecture*, December 1995, pp. 57–69.

[57] B. L. Deitrich and W. W. Hwu, "Speculative hedge: Regulating compile-time speculation against profile variations," in *Proceedings of the 29th International Symposium on Microarchitecture*, December 1996, pp. 70–79.

[58] 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*, May 1991, pp. 266–275.

VITA

Brian Lee Deitrich was born in Red Bank, New Jersey, in 1965. He pursued his undergraduate studies at the Rose-Hulman Institute of Technology in Terre Haute, Indiana, where he received the B.S. degree in electrical engineering in 1988. After receiving the B.S. degree, he went to work for Motorola's Government Electronics Group in Scottsdale, Arizona. While working for Motorola in 1994, he earned his M.S. degree in computer engineering from the National Technological University based in Fort Collins, Colorado. In the fall of 1994, he joined the Center for Reliable and High-Performance Computing as a member of the IMPACT project directed by Professor Wen-mei Hwu. After completing his Ph.D. work, he will join Motorola's Computing Systems Research Laboratory in Schaumburg, Illinois.