AUTOMATIC PROFILER-DRIVEN PROBABILISTIC COMPILER OPTIMIZATION

by

Daniel Coore

Submitted to the

DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

in partial fulfillment of the requirements

for the degrees of

BACHELOR OF SCIENCE

and

MASTER OF SCIENCE

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June, 1994

Signature of Author_____

Dept. of Electrical Engineering and Computer Science, May 15,1994

Certified by_____

Thomas Knight Jr., Principal Research Scientist, Thesis Supervisor

Certified by_____

Dr. Peter Oden, Company Supervisor, IBM

Accepted by_____

F. R. Morgenthaler, Chair, Department Committee on Graduate Students

Automatic Profiler-Driven Probabilistic Compiler Optimization

by

Daniel Coore

Submitted to the Department of Electrical Engineering and Computer Science
on May 15, 1994 in partial fulfillment of the
requirements for the Degrees of Bachelor of Science and Master of Science in
Computer Science

Abstract

We document the design and implementation of an extension to a compiler in the IBM XL family, to automatically use profiling information to enhance its optimizations. The novelty of this implementation is that the compiler and profiler are treated as two separate programs which communicate through a programmable interface rather than as one integrated system. We also present a new technique for improving the time to profile applications which resulted from the separate treatment of the compiler and profiler. We discuss the design issues involved in building the interface, and the advantages and disadvantages of this approach to building the feedback system.

As a demonstration of the viability of this implementation, branch prediction was implemented as one of the optimizations using the feedback mechanism. The benefits of the implementation are evident in the ease with which we are able to modify the criterion that the compiler uses to decide which type of branch to penalize.

We also address some of the theoretical issues inherent in the process of feedback-based optimization by presenting a framework for reasoning about the effectiveness of these transformations.

VI-A Company Supervisor: Dr. Peter Oden
Title: Manager, Compiler Optimization Group, Dept 552C

Thesis Supervisor: Thomas F. Knight Jr.
Title: Principal Research Scientist

Acknowledgements

Peter Oden, my supervisor, and Daniel Prener, a second supervisor of sorts, spent much time helping me with many of the technical problems I encountered in the implementation; without their patience and willingness to help, I would never have completed it.

Ravi Nair wrote the profiler and provided all the technical support for it that I needed for this thesis. He also contributed a major part to the efficient node profiling algorithm developed. Jonathan Brezin, as well as the rest of the RS6000 group, provided much support during my time at IBM.

Equally important was the support I received, while back at MIT, from the AI Lab Switzerland group. They readily accepted me as one of the group, and helped me prepare my oral presentation. Michael "Ziggy" Blair provided valuable insight into aspects of the subject matter that I had not considered before. Tom Knight provided useful guidance in the presentation of this thesis.

Throughout the development of this thesis, I have relied on many friends for support and guidance. Radhika Nagpal helped me define the thesis when all my ideas were still vying for my attention in the early stages. Bobby Desai provided precious time listening to my raw ideas, and helping me refine them—he perhaps, knew more about my thesis than anyone else. Sumeet Sandhu helped me endure through the long arduous hours and helped me keep my wits about me.

All my friends from home who were always interested to know the progress of my thesis helped me stay on track, Tanya Higgins in particular has my wholehearted appreciation for that. Last, but by no means least, I owe many thanks to my family; to whom I could turn when all else failed, and who have been very supportive throughout the entire endeavour. Finally, I offer a prayer of thanks to my God, the source of my inspiration.

Contents

List of Figures

Chapter 1

Introduction And Overview

1.1 Introduction

Traditionally, the transformations performed by optimizing compilers have been guided solely by heuristics. While this approach has had partial success in improving the running times of programs, it is clear that heuristics alone are not sufficient to allow us to get as close to optimal performance as we would like. What is needed is profiling to detect the *hot spots* of the program so that transformations can focus on these areas [1].

1.1.1 Background

In the majority of compiler implementations, if any profiling was used, it was done manually, as a separate process from compilation. The data from the profiling would then be used (manually) either to modify the source code, or to provide hints to the compiler so that it would do a better job. Ideally, we would like that feedback be provided automatically to the compiler and for it to be fully dynamic. The compiler should be able to compile code, profile the execution of the output, and then attempt to recompile based on the additional information from profiling. The whole process would probably have to be iterated over several times until a satisfactory proximity to optimality was achieved — even this state would not be a final one, since the application would be recompiled as its usage pattern changed.

The idea of automatically providing feedback to the compiler to improve the effects of its optimization phase has been attempted, but research in the area has been scant. Most of the past work has emerged in connection with research on VLIW machines, however with the decline in popularity of that architecture – particularly in industry – research in feedback based compiler optimizations has flagged commensurately.

Probably the earliest works involving feedback based compilation were done by Fisher [11]. He described *Trace Scheduling* as a technique, relying on information from a profiler, to determine paths through a program which were most likely to be executed sequentially. Ellis describes in detail [9], the implementation and application of *Trace Scheduling* to compilation techniques. Both Ellis' and Fisher's works were produced through investigative

efforts in VLIW architecture.

More recently, there have been other efforts at using feedback from profilers to direct compiler optimizations. The bulk of this effort has come from the University of Illinois where an optimizing compiler using profiler information has already been implemented and is currently being maintained [4]. Most of the background research for this thesis was centered around the research results of that compiler.

1.1.2 Implementations of Feedback Based Compilation

One of the important problems with feedback based optimization is that it is inherently expensive both in time and space. First, the need to profile when tuning is desired means that compilation time will now be dependent upon the running time of the programs compiled[1]. The space problems arise because of the need to constantly recompile applications. If a fully dynamic system is to be realized, the compiler must be readily accessible—probably lying dormant within the runtime system, but ready to be invoked whenever it is deemed necessary to recompile an application. In a sense, the compilation process is distributed over much longer periods which means that data structures have to be maintained between optimization attempts, thereby increasing the space required for the runtime system and the whole compilation process in general. Under this ideal view, compilation is an ongoing process and continues for as long as the application is being used. While there are no implementations of such a system, there are many of the opinion that it is an attainable ideal [8], and in fact, one that represents the next generation of compiler and runtime-system interaction [3].

In all the implementations documented, the system implemented was a compromise between the way compilation is currently done and the ideal described above. Typically, there was a database per application which was updated everytime the application was profiled. The compiler was called manually whenever there was a chance for improvement in the application's performance. The profiler was called automatically from the compiler, and the recompilation using the additional profiled data was also done automatically.

Given the computing power available today, this implementation is a reasonable compromise since our present technology lacks the ability of self-evaluation that the compiler would need to be able to automatically determine when recompilation was advisable. By automating the compiler-profiler interaction, much of the tedium is expedited automatically. At the same time, maintaining manual control over the frequency of compilation ensures that the runtime system is not spending too much time trying to make programs better without actually using them.

1.1.3 Motivations

The idea of using feedback to a compiler to improve its optimization strategy has never been very popular in industrial compilers and has remained mostly within the academic research domain. The IBM XL family of compilers is not an exception, and we would like to explore

[1] Profilers usually limit the amount of data collected. So there will be an upper bound on the running time, though it may be quite high.

the costs and benefits of implementing such a feedback system within the framework already present for these compilers.

In light of rapidly increasing size and scope of application programs and the constant demand for efficiency, coupled with ever increasing processor speeds mollifying concerns about the time investment required, we feel that it is worthwhile researching this approach to compiler optimization. Also, from an academic standpoint, the large space of unexplored problems in this area promise to provide exciting research with highly applicable results.

1.1.4 Objectives

This thesis will describe the design and implementation of such a feedback based compiler system with a specific focus on the RS6000 super scalar architecture. We hope also to address some theoretical issues inherent in the process which are still largely unexplored.

The primary focus will be on the design issues involved in the implementation of the feedback mechanism within the XL family of compilers. The secondary goal is a two-pronged endeavour: a theoretical approach to reasoning about the effectiveness of feedback-based optimizations, and an exploration of the ways in which feedback information can be applied.

1.1.5 Features of This Implementation

One important difference between the goals of this thesis and the implementations aforementioned is that the compiler and the profiler are considered as separate tools, which can be enhanced separately as the need arises. In the past implementations, the profiler was incorporated into the compiler and the two functioned as a single program. While there are certain advantages to this type of implementation such as greater efficiency, the compiler is restricted to receiving only one type of feedback information — whatever the embedded profiler happens to return. Having the option to easily change the profiling tool allows greater flexibility in the type of information fed back, and can for example, be invaluable in investigating how changes in program structure affect various run-time observables.

Another difference in this implementation is that this work was an extension of an existing compiler to allow it to take advantage of the additional information (from profiling). This fact meant that there were several imposed constraints that restricted the solutions to some problems that would otherwise have been solved by the past implementations. For example, the order in which optimizations were done was critical in the compiler, and this restricted the points in the optimization phase at which the feedback data could be used.

Finally, we would like to point out that this implementation does not attempt to achieve the ideal setting of a dynamic compiler constantly recompiling applications as the need arises. It is still invoked manually, as the user sees the need for improvement in the code, but it does perform all the profiler interaction automatically. While this approach is not ideal, we feel that the results that this research yields will provide important groundwork for implementing the ideal environment described earlier.

Theoretical Work

Although much work has been done to show the experimental success of feedback based optimizations, those experiments were performed under very controlled environments. Typically the results of those experiments, and their ramifications, were specific to the particular programs compiled. These results were often used to cite the benefits of the particular optimization under investigation, but we would like to be able to show with more confidence that the optimization is valuable in general (i.e. "on average") rather than merely for the programs that happened to be in the test suite. Although attempts have been made to theoretically model the behaviour of feedback-based optimizations, no models have been able to predict their stability nor even evaluate the suitability of an input as a *predicting input*[2] to the application.

While the design and analysis of such a model in its full specification is an ambitious task, one of the goals of this thesis was to address the issues involved with that task. Although the model developed is incomplete in that its accuracy has not been measured, we present some of the necessary features of any such model and put forward arguments for why the model is reasonable. We have presented the model, despite its incompleteness, in the hope that it will provide essential groundwork for furthering our understanding of the subject matter. Any progress in this direction will help to solidify cost-benefit analyses and will certainly further our understanding of program-input interaction and the relationships between program structure and program performance.

1.2 Overview of The Implementation

The feedback system was designed to allow the most naive of users to be able to use it without having to worry about issues that were not present before the feedback mechanism was introduced. At the same time we wanted to make it powerful enough to allow much flexibility in the manipulation of the statistical data without exposing more details of the compiler than were necessary. The few aspects of the feedback mechanism which require user interaction are hidden from the user who is not interested in them, through the use of suitable defaults.

In this manner, the user who simply wants the benefit of feedback in optimization need not worry about the type of profiler being used or any of the other issues that are not directly related to the application itself. At the same time, another user who wanted to see the effects on a particular optimization when decisions are based on different types of statistics (which may need different profilers to provide them) could specify this in a relatively painless manner without having to know anything about the internals of the compiler. Note however, that the second user would have to know more about the system than the first, in order to be able to change the defaults as appropriate. The general idea is to provide suitable layers of abstraction in the system that expose only as much detail as is desired by users without compromising the benefits of feedback based compilation.

[2] A predicting input is one which is used with the application at profile time to provide statistics that may guide the optimizations later.

1.2.1 Compiling With Feedback

There are two phases to compilation. The high-level decomposition of the whole process is as follows:

1. Phase 1
   (a) Generate unoptimized executable code from source
   (b) Construct the corresponding control flowgraph
2. Phase 2
   (a) Profile the unoptimized code from phase 1
   (b) Annotate the control flowgraph with the statistics
   (c) Generate optimized code based on both feedback information and on traditional heuristics

All this is abstracted away from the user through a high level program which manages the processing of the two phases without input from the user. This high level program is the vehicle through which the user interface is presented since this is the only program that the user needs to be aware of to use the feedback mechanism.

1.2.2 Unoptimized Code generation

The first phase is mainly concerned with generating executable code in a manner that preserves some kind of context about the instructions so that when the profiled data is returned, the compiler knows to which instructions the statistics pertain. Since the compiler is called twice, it must be able, in the second phase, to reproduce the flowgraph generated in the first phase, if it is to perform different operations on the output code in both phases with any level of consistency between the phases.

To achieve that consistency, the code is largely unoptimized so that the second phase will generate the same flowgraph as the first phase does. Were the code optimized within the first phase, then the profiler, which looks at the object code produced, would provide data on instructions that the compiler would have difficulty tracking the source of. This difficulty arises because of the extensive transformations that occur when optimization is turned on within the compiler. Therefore, when the profiler returned a frequency count for a particular basic block, the compiler would not necessarily be able to figure out the connections between the flowgraph at the early stages of compilation and that basic block. The alternative would have been to have the compiler fully optimize in both passes, so that the flowgraphs would be consistent between phases, but unfortunately the implementation of the compiler does not allow certain optimizations to be repeated. Therefore, if the code were fully optimized before the statistics from profiling could be obtained, the code would not be amenable to several of the possible transformations that take advantage of the feedback data.

Actually, the requirement for producing a consistent mapping from profiled data to data structure is less stringent than pointed out above. We really only need to capture the feedback data at some point in the optimization phase, and then introduce that data at the same

point in the second phase (see Chapter 2). The reason for this particular implementation strategy is that the compiler assumed would attempt to do all the optimizations it knew about if optimizations were turned on. Only if they had been disabled from the command line, would they be ignored. This user directed choice of optimization, while a good idea in general, caused a problem in identifying a point in the optimization process that could be repeatedly reached with the same state on every pass. The problem is that once a point in the process has been chosen, we want to turn off all the optimizations that come after that point, and go straight to the next phase of compilation (register allocation in this iplementation). However, a simple jump to the next stage is not good enough because there were some processes that had to always be done, but which followed optimizations that were conditionally done. Therefore, the solution would have been to find all the essential subroutines that had to be executed and then ensure that no matter which point was chosen to capture the fedbakc data, all those subroutines would get executed. It was difficult, though not impossible, to separate these subroutines from the conditional optimizations, and so we opted for the extreme choices of all or no optimizations.

It turned out that there were a few optimizations that could be performed early and were repeatable and so could be done in the first phase. Ideally we would have liked to optimize the code as much as possible, while observing all the constraints imposed. The executable generated from the first phase has to be profiled on several inputs, which means that it must be executed for each of those inputs. The profiling process is the bottleneck in the whole compiilation process and so it is highly advantageous to have this code run as fast as possible.

Perhaps the biggest drawback to having the profiler and compiler separated is that there is no *a priori* means of communication between the two tools. Specifically, there is no way for the compiler to associate the frequency data returned by the profiler with the instructions that it generates. Typically the profiler returns the addresses of the instructions along with the frequencies of execution, but this is useless to the compiler since the addresses were assigned at bind (link) time which happens after compilation is done. The solution used by this implementation was to use the space in the object format for line numbers to store unique tags per basic block. The profiler could return the line number (actually what it thought was the line number) with the instruction frequencies and the compiler would use that "line number" to decipher which instruction was being referenced and hence figure out its execution frequency. This solution has the advantage that many profilers will already have the ability to return line numbers along with instructions so that the compiler is still easy to be compatible with. The disadvantage is that the association set up with this scheme is rather fragile, and so does not make keeping databases of profiled runs feasible since minor changes in the source will cause a complete breakdown of the tag to basic block associations.

1.2.3 Flowgraph Construction

Compilation is done on a per file basis which is further broken down on a per procedure basis. A flowgraph is built for each procedure in the file. Originally, only one flowgraph was constructed at a time, and was processed all the way to code generation before processing the next procedure. In the modified implementation, we constructed all the flowgraphs for

the procedures at once, so that when the statistical data was read in the second phase, only one pass of the data file would be necessary. This approach is more efficient in terms of time since the interaction with the file system is reduced to a minimum, however there is a corresponding space tradeoff since all the flowgraphs have to be contained in memory at once. In practice, this memory constraint does not prove to be a problem because files are typically written to contain only one major procedure. It is extremely rare that files contain several large procedures.

One of the types of information that is used by most transformations is the frequencies of flow control path traversals (edge frequencies). However, this information can be expensive to obtain if the profiler does not have any more information than the program and inputs it is given to profile. In previous implementations where the compiler and profiler are integrated, the profiler can analyze the flowgraph and then decide how to optimally profile it to obtain all the desired information. This feature is lost in this implementation since the profiler and compiler are separated, however the edge frequency information is still as important. The solution was to require that the profiler be able to produce frequency counts for basic blocks — the nodes of the control flowgraph — and place no constraints on its ability to produce edge information. This requirement also had the side effect of making compatibility easier. In order to obtain the edge frequencies solely from the node frequencies, the compiler had to modify the flowgraphs so that there would be enough node information returned by the profiler to solve for the edge frequencies. The modification was done according to a newly developed algorithm which is based on the fact that control flow obeys Kirchoff's current law at the nodes of the flowgraph. The algorithm modifies the flowgraph optimally in the sense that the minimal number of edges are instrumented, and it works on arbitrary topologies of flowgraphs.

Since in this stage, no data is available to actually solve for the edges, each edge is annotated with a symbolic expression which represents a solution in terms of the node frequencies and possibly other edge frequencies, which are all determined immediately after profiling. After the profiling is done, the actual numbers are substituted into the expressions which are then evaluated to yield the frequencies for the edges. The details of the implementation and the algorithm including arguments for its correctness and optimality are given in Chapter 3, however the details therein are not necessary for appreciating the practicality of the algorithm.

1.2.4 Profiling The Unoptimized Code

The interaction between the compiler and profiler is summarized in a *specification file.* This file contains all the templates for reading the output of the profiler, and also all the instructions for invoking the profiler. This file is actually provided by the user, but by producing default ones for popular profilers, casual users need not have to worry about its details.

The other requirement for profiling the code is the data set on which the code should be executed while it is being profiled. This is provided through yet another file which contains all the inputs to the application that the user would like the code to be profiled with. Each line of this input file contains exactly the sequence of characters that would come after the invocation of the program under manual control. Although all users must specify this file

and create it themselves, it is a relatively painless process, since the data that is put into this file is no more than the user would have had to know about in any event. The data presented within this file is called the *predicting input set* since these are the inputs which ultimately get used as a basis for assigning probabilities to each branch of the flowgraph. Thus they "predict" which paths through the flowgraph are likely.

1.2.5 Annotating The Flowgraph

The profiler produces one output file per profiled run which is done once per input in the input file. Therefore, at the end of the profiling stage, there are as many profiler output files as there are inputs in the input set. The frequencies for the basic blocks, produced in each file, are accumulated and then associated with the appropriate tag for the basic block (the tag was produced within the file along with the basic block it was associated with). These accumulated values are then used to annotate the appropriate node of the appropriate flowgraph — as dictated by the tag. The final step in the annotation process is to substitute the actual accumulated values into the expressions computed for the edge frequencies in the flowgraph construction stage to yield the values for the accumulated edge frequencies of the flowgraph.

1.2.6 Optimized Code Generation

Although there are several different optimizations that are possible with feedback information available (see Chapter 4), we had enough time to implement only one of them — namely branch prediction. Even so, it was only implemented for handling the limited case of an occurrence of an if-then-else construct. This optimization was heavily machine dependent and will only be beneficial to machines with uneven penalties for conditional branch paths. This optimization was apt for the RS6000 which has a branch penalty of up to three cycles for conditional branches taken and zero cycles for conditional branches not taken; unconditional branches have zero penalty provided there are at least three instructions before the next branch at the unconditional branch target address.

The compiler assigned probabilities to each edge based on the frequencies measured from the flowgraph annotation stage. Each edge was also assigned a cost dependent upon the inherent cost of traversing that edge and upon the number of instructions within the basic block to which the edge led. From these two quantitites, a weighted cost was computed and was compared with that of the other edge leaving the same node. If the edge with the higher weighted cost was sufficiently high (that is, above the other by some fixed threshold) then the compiler ensured that that branch corresponded to the false condition of the predicate of the conditional branch instruction. In this manner, edges with larger latencies and smaller probabilities are penalized more than those with smaller latencies and larger probabilities.

1.3 Summary and Future Work

In our paradigm for the user interface, we wanted to go even further and provide a means of abstractly specifying optimizations in terms of the transformations they would represent

on the control flowgraph. This would have been achieved by developing a language for manipulating the internal data structures of the compiler that could simultaneously refer to data being produced by the profiler. This would have allowed yet another level of user — namely the compiler researcher — a convenient abstraction to the compiler. Already there are two extremes of users who are accommodated: the naive user who is not concerned with the details of the profiler being used, and the performance-driven user who is constantly attempting to find the most suitable type of statistics for yielding the best improvement in code performance. The third type of user would have been the user who not only was interested in the type of statistics being used, but also with what was being done with the statistics.

This extension to the compiler interface is a major undertaking however, and could not be accomplished within the time frame set out for this thesis. An implementation of such an interface would prove to be extremely useful since it would not be necessary for someone to understand the internals of the compiler before being able to specify an optimization.

Numerous other optimizations could have been implemented as they have been in the other implementations of feedback-based systems. For example, the entire family of trace based optimizations would make excellent use of the feedback information and would have probably yielded favourable performance enhancements. These optimizations, although highly applicable were not focussed upon since their effects have been widely investigated with positive results. Other optimizations which could not be completed in time, but which would be interesting to investigate include applications of feedback to code specialization and determining dynamic leaf procedures[3].

1.3.1 Summary

Although rudimentary in its implementation, and lacking many of the interesting uses of feedback data, the feedback based compiler has provided useful information about dealing with feedback systems and has raised many issues to be further researched. This thesis established a modular implementation of a compiler with a feedback mechanism to guide its optimizations. The knowledge obtained from tackling the design problems that arose in the implementation will provide useful guidelines to future implementations. One of the useful results is that the expense of profiling arbitrary programs can be reduced by reducing the amount of instrumentation that would normally be done. This reduced cost of profiling enhanced the viability of a modular implementation since under normal circumstances the profiler would not have information available in integrated implementations and would therefore be expensive to run in the general case. We have also developed a framework for assessing the effect of optimizations on programs using feedback information.

1.4 Bibliographic Notes

Aho, Sethi and Ullman state that we can get the most payoff for the least effort by using profiling information to identify the most frequently executed parts ofa program [1]. They

[3] A leaf procedure is one that does not call another procedure, it is a leaf in a call graph for the program.

advocate using a profiler to guide optimizations, although they do not make any mention of automating the process.

At MIT, members of the Transit Project group and Blair [8, 3] independently advocate the adaptive dynamic feedback-based compiler as the archetype compiler of the future.

In the Coordinated Science Laboratory at University of Illinois researchers have implemented IMPACT [4] a feedback-based optimizing compiler that uses the feedback to implement several different optimizations ranging from trace scheduling with classic optimizations to function inlining.

Fisher first came up with trace scheduling in 1981 [10] as a means of reducing the length of instructions given to a VLIW machine. Ellis later describes in detail how this is done in his PhD thesis under Fisher [9].

Chapter 2

Design and Implementation

This chapter describes the details of the implementation and discusses the design issues that were involved. The first section describes in detail the general requirements of the compiler, profiler and the interface for them to cooperate in a mechanism for providing feedback to the compiler. The second section describes the possible implementations for the general requirements outlined in the first section, the actual implementation that was used and the reasons for choosing it. In many cases, a problem was solved in a particular way because it was the best solution in terms of accommodating the constraints imposed by the conventions of the existing tools and environment. In those cases, we try to point out what might have been ideal and any major disadvantages of the chosen implementation.

As mentioned before, we chose to implement the profiler and the compiler as two separate tools for two main reasons. The first is that the system is more modular in this way, because now the compiler and the profiler can be upgraded individually so long as the protocol with their interface is not violated; therefore, maintenance is simplified. The second reason is that the compiler can be easily configured to optimize according to different types of statistics. For example, cache miss data per basic block or per instruction would provide guidance to the compiler that may help in laying out the code to improve cache behaviour. The flexibility offered by the modularity provides a powerful research tool which would be much more difficult to implement and manipulate with an integrated compiler-profiler system.

2.1 Designing The Feedback Mechanism

The first step in the design of the interface was to identify the necessary properties of the compiler and the profiler for them to be able to interact in a meaningful manner. The next step was to decide how the dynamic data (from the profiler) should be captured, what should be collected and how it should be stored. In other words, the first consideration was a question of how the data should be obtained and the second was a matter of what the collected data could be used for.

2.1.1 Necessary Properties for Feedback

The compiler produces executable code as input to the profiler, and it also reads in the output of the profiler as part of its own input. Therefore, the compiler must be able to produce an executable suitable for the profiler and also be able to parse the output of the profiler. Suiting the needs of the profiler may be done in several ways according to the profiler being used. Parsing the output of the profiler is again highly dependent on the profiler being used. Even beyond these problems of protocol specification, the compiler needs to be able to identify the instructions being referred to by the profiler when it returns its frequency counts. Summarizing, there are three important properties that the feedback-based compiler must have:

1. It must be able to produce code suitable for the profiler.
2. It must understand the format of the profiler output.
3. The compiler must understand how the profiler refers to the instructions that are presented to it for profiling.

To elaborate the point, since the compiler and the profiler are two separate programs, they do not share any internal data structures. Therefore, the compiler's view of the code generated is very different from the profiler's view of the code executed. In particular, the compiler never sees the absolute addresses assigned to the instructions, whereas that is all the profiler has to refer to the instructions by. Furthermore, the data actually desired is basic block counts, so the compiler has the additional problem of identifying the basic block to which a particular instruction in the final output belonged.

We mention that we desire basic block counts although profilers can usually return frequency data on many granularities of code ranging from machine operations to basic blocks. Basic block information is preferred because that granularity is coarse enough to provide a good summary of control flow from the data, and fine enough to allow information on finer granularities to be obtained. For example, knowing the frequency of execution of all the basic blocks, along with the instructions that are in the blocks, is sufficient to deduce the instruction distribution. Each instruction within a basic block is executed as many times as the basic block itself is, so by summing the frequencies for each basic block in which a particular type of instruction appears, we can obtain the total number of times that instruction type was executed.

2.1.2 Types of Data to Collect

Deciding what kind of data to store is largely determined by the features of the profiler. In most cases, the profiler will give only various types of frequency summaries for some granularity of code — basic block, instruction, or operation for example. For these, deciding what to store is not much of a problem since we can typically store all the frequencies. However, if the profiler has the ability to return many possible types of summaries it may be too expensive to store all the possibilities. For example, one could imagine a profiler that had the ability to return information about the values of variables within the basic block,

so that at the end of the profiled run, it would also return a distribution of the range of values assumed by the variables within a block. This type of information would be rather voluminous to store especially if it were done for several variables across several basic blocks and for an especially long profiled run. In cases like these, decisions have to be made (by the designer) about what kinds of data are relevant for the uses he has in mind.

Therefore, deciding what the feedback data will be used for is highly dependent on the kind of data that the profiler can return, and conversely, deciding what kinds of data to store — in the cases where we have a choice — is largely dependent on the proposed uses of the data. This interdependence seems to suggest that the data stored should, in some way, be linked to a semantic model of the program, since the semantics will be the indicators of what kinds of optimizations can be done in the later phases of compilation. Also, the semantic model used should be one that the compiler is already familiar with (or will be eventually) since it represents what the compiler "knows" about the program. By annotating this model with the feedback data, the compiler will — in some sense — be able to ascribe meaning to the data.

Having data that does not directly relate to any of the compiler's semantic models will probably be difficult to use in the optimization phase, since those statistics will be hard to describe in terms of the model that the optimization is implemented in. For example, if the semantic model for a particular optimization was the control flowgraph, and the data that the profiler returned included the frequencies of procedures called, then that data would not be as effective as if the semantic representation had been a call graph instead. In that example, if there was no consideration ever given to the call graph in the optimization phase of compilation, then it was probably not worth the space or the time to collect that particular type of data.

We see that there is an interdependence between the data that is available in the first place, the data that is stored and the possible uses of it. In some sense, the semantic models that are available determine what kinds of data it makes sense to store, and also what might be done with the data by the optimizer. Of course, the programmer may decide that he wants to use some particular type of data for a particular optimization, but if there is no semantic model to support the data directly, then he must implement a model which can take advantage of the data he is interested in using. Not surprisingly, both the kind of data being returned and the intended uses for the data will affect when and how, during the compilation process, the data is captured.

2.1.3 When to Collect the Data

When the data ought to be captured is determined by properties of the compiler and by the intended uses for that data. How the feedback data is captured also depends on the compiler. In general, code generation has to occur at least twice. Once to give the profiler code to profile, and the second time to use the feedback data with the optimizer. How these two passes of the code generator are done is subject to two general constraints:

1. The code generated on the first pass must be done using only the state of the semantic models at the time of data capture.

2. After the point of data capture, all of the semantic models for which data will be collected must either be exactly reproducible if necessary or remain intact after the first pass of code was generated.

The first constraint guarantees that the code to be optimized using feedback is the same as what was profiled. The second ensures that the data retrieved is associated with the semantic models in the appropriate way since any assumptions about the code in the first pass of code generation will not be violated afterwards. When the profiler results are returned, they ought to be used to annotate the semantic models present after that data has been retrieved. In the case when this pass is a completely different run from the first pass of the code generator, the second constraint ensures that the semantic model that is annotated with the data will be generated to have the same structure as the semantic data structure that produced the code that was profiled. The first condition guarantees us that that data structure is in fact the one that corresponds to the code that was produced in the first pass and profiled. In the case when the semantic models are preserved across the first pass of code generation, the first constraint again ensures that the data returned corresponds with the models.

Since the code produced will be profiled for each of the inputs given in the input set, it should be optimized as much as possible. In this way, the overall profiling time will decrease. There is a tradeoff involved, though that should be noted. There are many optimizations that we would like to use in conjunction with feedback information such as loop unrolling, and common subexpression elimination [1] with large traces obtained from application of the feedback data. Therefore, it would seem reasonable to collect the feedback data before these optimizations took place. However, we cannot use them in the code that will be profiled unless we are willing to repeat them. While repeating those optimizations might not be a bad idea in general, there is the question of whether the optimizations will adversely affect each other when they are used in the second pass of code generation.

The tradeoff then, is whether to try to produce code that can be profiled quickly, or to sacrifice time in the profiling stage of the compilation process, in the hope that the resulting optimized code will be much faster than otherwise. The two options indicate collecting the feedback data at two different possible extremes: either very early when there is no first pass optimization and therefore greater opportunity for application of feedback data or very late, when the output code will be very fast, but the feedback data will not be used very much to improve the code generated. It seems that duplicating the optimizations so that we get both benefits is the most favourable option. Failing that however, choosing some suitable compromise in optimization of the profiled code seems to be the correct action to take. A judicious choice can be made by using some of the optimizations that do not stand to gain much from feedback information, such as constant folding and propagation on the code in the first pass of code generation.

2.2 The Implementation

This section discusses the possible implementations given the design requirements discussed in the first section, and gives justification for the implementation chosen. In many instances, the chosen implementation was dictated by the constraints of the already existing system

rather than any deliberate decision. For these, we discuss how the implementation could possibly be improved if these constraints were relaxed.

2.2.1 The Compiler

The compiler used was IBM's production compiler called TOBEY[1]. It is written in PL.8, a subset of PL/1, which was developed several years ago explicitly for implementing an optimizing compiler for the then new RISC architecture. The TOBEY compiler has had several optimizations implemented in it throughout its lifetime, resulting in a highly optimizing compiler; however, it has also grown to immense proportions, with many esoteric intricacies that often proved to be obstacles to successful code manipulation.

The compiler's operations are broken down into five stages. The first stage includes all the lexing and parsing necessary to build the data structures that are used in the subsequent stages. The second stage generates code in an intermediate language called XIL, a register transfer language that has been highly attuned to the RS6000 architecture. The third stage performs optimizations on the XIL produced from the second stage and also produces XIL. Register allocation is done in the fourth stage and further optimizations are then performed on the resulting code. The fifth and final stage generates assembly code from the XIL produced in the fourth stage.

There are essentially two levels of optimizations possible within the compiler, the actual level to be used for a particular compilation is given as a compile-time parameter passed with the call to the compiler. At the lower level of optimization, only commoning is done and only within basic blocks. The higher level of optimization implements all the classic code optimizations such as common subexpression elimination and loop unrolling as well as a few which were developed at IBM such as reassociation [5]. In this implementation, there is actually a third level, not much higher than the lower one, which performs minor optimizations within basic blocks such as value numbering and dead code elimination. This was the level at which code was generated for profiling since all the optimizations at this level satisfied all the constraints mentioned earlier.

One unfortunate characteristic of the optimizations done within the TOBEY compiler is that the operation of some optimizations are not independent of others. Therefore, the XIL produced by some optimizations is not always suitable as input to those same optimizations and to others. The reason this is unfortunate is that the order in which optimizations are done is very important and requires detailed knowledge of the interdependencies to be safely modified. Two main problems arose in the implementation of the feedback mechanism because of this: modifications to the code to accommodate the data structures for the mechanism were often difficult or cumbersome, and some optimizations that stood to benefit from feedback information could not be done on the code that was to be profiled.

2.2.2 Producing Code for the Profiler

In this implementation, the code was generated twice in two separate runs of the whole compiler. The profiler was given only the code from the first pass, and this code had minimal

[1]Toronto Optimizing Back End with Yorktown

optimizations performed on it. On the second pass, all the data structures constructed for code generation in the first pass were duplicated exactly, just as was discussed in the constraints for data collection and code generation. Since the compiler is deterministic, the code structure produced is exactly the same as that of the first pass, and hence the same as that given to the profiler. Therefore, any reference to the data structures for the code (say a control flow dependency graph) of the first pass is valid and has the same meaning as the equivalent reference in the second pass.

We could not attempt the preferable option of duplicating the optimizations used between the two code generation passes because of the incompatibility of some optimizations with others as mentioned above. Within the TOBEY compiler, there are dependencies between the optimizations themselves, so that some cannot be performed after others, and some cannot be repeated on the same XIL code. These constraints prevented the first pass of code from being optimized too highly.

2.2.3 Reading Profiler Output

As mentioned in the earlier section, the second requirement of the compiler is that it needs to know the format of the output of the profiler so that it can automatically extract the appropriate statistics from the profiler's output. This requirement is essentially for automation only. The first requirement would have been necessary even if the data was going to be fed back to the compiler manually. Automating the feedback process requires that the compiler be able to identify important portions of the profiler output and ascribe the appropriate meaning to them. For example, the compiler must know where in the output to find the frequency count for a particular basic block.

As part of the understanding that the compiler has of the profiler's output, the compiler must know not only where to find the appropriate data, but also what ought to be done with that data. For example, it may be that the profiler produces frequency counts as well as the length of the basic block in terms of instructions. If we care about both items of information, then the compiler needs to know not only where to find these items in the whole output, but also how these data should be used in conjunction with its semantic models. This particular detail posed a small problem in the interface since it was not obvious how the user could describe what data was supposed to be interesting and how it would be interesting, without having to know too much of the compiler's innards. The solution was to settle for the fact that a control flowgraph could safely be assumed to be used by the compiler. Then any data was considered to be for annotation of that flowgraph. Ultimately, we would like something more universal than this, but that would require a sophisticated means of specifying the semantic model to use and further, the way in which the data should be used with that model.

2.2.4 Identifying Profiler References

The third requirement of the compiler is that there must be some means of identifying the instructions (and the basic blocks) that the profiler refers to when it returns its statistics. The solution employed was to generate a tag per basic block which was unique within a file. This tag was then written out with the rest of the object code into the object files, in such

a way that it would be preserved throughout linking and loading and finally dumped into the executable where it would be visible to the profiler. The profiler would then report the tag along with the frequency for the instruction associated with the tag.

There were two potential problems with this proposed solution. The first was that while the format of the object file was rather liberal and would allow such tags within it, there was no simple way to force the linker and loader to reproduce those tags within the executable since the format of the executable, XCOFF[2], is quite rigid, and the linker discards unreferenced items. The second problem was that since this data inserted by the compiler was custom data within the executable, the average profiler would not have been capable of returning that data — defeating the central goal of wide compatibility.

The solution to both problems was to use the line number fields within the executable to contain the tags. Since the XCOFF format already had space to accommodate line numbers, no additional modification was necessary, either to the XCOFF format or to the linker and loader, to preserve the tags from the object files. Also, since the line number is a standard part of the XCOFF format, the chances that the profiler will be able to return that data field are high. The limitations of this solution are that the tags are restricted to being at most 16 bits wide which turned out to be suboptimal, and that the line number information, which is very useful while debugging, is unavailable while the tags are being reported. That second limitation is not a problem in general since on the second pass, we can restore the actual line numbers to the fields. In the second pass, there is no need to have profiler tags since the compiler will not be profiling the output of the second phase. Were the compiler to continually massage the same code over and over by repeatedly profiling and recompiling, then we would be forced to permanently substitute tags for line numbers, in which case an alternate solution would probably have to be found.

2.2.5 Interfacing with the Compiler

Designing the interface between the compiler and the profiler is, perhaps, the most important part of the entire project. The specification of the interface determines the kinds of profilers that are compatible with the compiler, and what features, if any, of the compiler will need to be preserved when it is upgraded. A second aspect of the interface was that between the compiler and the user. This aspect allows the user to be able to specify the manner in which the feedback data should be processed as well as indicate to the compiler which parts of the profiler data should be considered important.

An integrated user/profiler interface was developed because of the close interrelation between their functions. It should be noted however, that the user interface aspect was not stressed. So while it provides the user with means of expressing manipulations on the feedback data, it could only offer modest expressive power, and an admittedly esoteric syntax for using it.

All the programmable aspects of the interface were specified in one file called the *specification file*. This file contained all the directives to the compiler to allow it to communicate with the profiler according to the requirements outlined above. In particular, the specification file contained information on how the compiler should call the profiler, parse its output and

[2]IBM's extension to the UNIX standard C FF format [13]

also choose which parts of the profiler output were important.

One disadvantage to incorporating this type of generality in the interface is that there is a danger of escalating the complexity to such levels that even simple tasks become tedious or difficult to express. To alleviate this problem, a specification file was designed for the particular profiler being used. One of the input parameters to the compiler was the name of the specification file to use in interfacing with the profiler. This particular specification file was set up as the default, so that a simple call to the compiler with feedback turned on would use the default profiler, without the user having to worry about it. One could envision several specification files being implemented, one for each popular profiler available, so that many different types of feedback might be available without the user having any detailed knowledge of the system. At the same time, the compiler researcher who may care to investigate the effects of different types of feedback information available from the same profiler would want the expressive power offered by the programmable interface to allow him to pursue his research. We feel that this implementation of the interface provides a suitable compromise between simplification of use and expressive power available.

2.3 The Profiler

Almost any profiler could be used as part of a feedback mechanism, and so it would not be very instructive to try to identify essential features of a profiler for this purpose. However, there are several restrictions on the profiler that can be used with this implementation of the feedback mechanism, and this we discuss in detail below.

Since a central goal of this implementation was for the system to be able to use the results of many profilers, the requirements on them were kept to a minimum. The essential requirements of any profiler compatible with the system are three:

1. It must be able to produce frequency counts for basic blocks.
2. It must be able to return the source line number corresponding to an instruction in the executable.
3. It must return along with that line number, the source file name.

The first requirement is necessary — even if that data will not be used for optimizations — because the algorithm that determines edge frequencies depends on these values. Although, a limitation of sorts, it is not very serious, since in most instances, this piece of information is exactly what an optimization is looking for. The real limiting factor of this requirement is that the exact counts are needed as opposed to a program counter sampled profile of the program. This immediately excludes a large class of profilers from being compatible, although this limitation is mitigated by the fact that profilers of that class typically work at a coarser granularity than the basic block. Most of those profilers—*prof* and *gprof* are examples—report sampled timing information on procedures and so tend to provide useful information for annotating a call graph, rather than a control flowgraph.

The ideal situation is for a profiler to be able to return both types of frequency counts and have both the call graph and control flowgraph as views of the program at two different

granularities. The call graph which would be annotated by the sampling profiles would help direct the basic block profiler by indicating the *hot* procedures. This feature would be extremely important in an adaptive dynamic implementation of feedback based optimization, but in this static setting, profiling at the coarser level is left up to the user.

2.3.1 The *xprof* Profiler

The profiler used with this implementation was the *xprof* profiler, designed and implemented by Nair [14]. One important feature of this profiler was that it did not require the instrumentation of nodes to be done by the compiler. That is, unlike *prof* and *gprof*, there was no parameter that needed to be passed to the compiler to allow the output to be compatible with *xprof*. This profiler could return various types of frequency information ranging from instruction type summaries to basic block frequencies. The primary mode of usage in the implementation for this thesis was for it to return basic block frequencies.

Along with the frequency of execution of a basic block, and its tag (line number), the *xprof* profiler could also return the start and end addresses of the basic blocks it encountered. These two values together indicated the size of the basic block in terms of instructions, which was used to help weight the branch proabilities in the implementation of branch prediction (see Chapter 4.

2.4 Bibliographic Notes

gprof and *prof* are profilers that are part of the UNIX standard. They both sample the machine state periodically and report a summary of the relative frequencies of the procedures that were executed. In contrast, *xprof* returns actual basic block counts for all the procedures executed.

Reassociation was developed by John Cocke and Peter Markstein at IBM [5] and was a more general technique than the similar, independent implementation by Scarborough and Kolsky [17].

Chapter 3

Efficient Profiling

3.1 Introduction

In this chapter we describe how the application being compiled was processed so that:

1. profiling the output code could be done efficiently,
2. the data returned contained enough frequency information to allow all the control flow statistics to be determined.

Proceesing the application in this way has two direct advantages. First, more profilers are compatible with the compiler since a profiler need not be able to instrument all the jumps and branches within a program to be compatible. Second, performance will be improved for most profilers that are capable of providing all the control flow statistics, since they typically provide that information by instrumenting all jump and branch instructions. In this implementation there are fewer instrumentation points and so we expect that the profiling process will be faster overall.

3.1.1 Description of the Problem

Profilers give us frequency information about the execution of a program on a particular input. The type of frequency information returned varies widely with profilers and can range from estimates on the time spent within particular basic blocks to an actual tally of the number of times a particular instruction was executed.

While a frequency count of each instruction executed is extremely useful for pointing out interesting parts of a program (like hotspots), it is often not enough for some types of analysis. Typically we not only want to know the number of times an instruction was executed, but we also want to know the context in which that instruction was executed, and we want statistics on that context. To illustrate the point, imagine we had a bounds checking routine that called the same error handling routine with different arguments for different errors. If we were curious to know which particular error happened most frequently,

it would be insufficient to know only the total execution tally of the error handling routine. We would need to know the number of times control was transferred from the point of error detection to the error handling routine.

This kind of frequency information for the context of instruction execution is usually available as a standard part of most profilers, however there is usually a substantial price in performance to pay for the added information. For example the *xprof*[1] profiler [14] suffers up to a 1500% performance decrease when keeping additional statistics for the control flow between the nodes executed. This performance degradation comes mainly from the instrumentation involved with each branch instruction. Instrumentation is the process of modifying a part of the original program so that tally information is automatically updated everytime that code fragment is executed. Instrumentation usually involves inserting a branch instruction before the code fragment under investigation. The target of this branch instruction contains some code to increment the counter for that code fragment, followed by a branch instruction back to the code fragment. Hence to obtain all the context statistics (edge frequencies), the instrumented code is larger and usually slower than the original code since the modified code now involves extra computation (including at least one other branch) per branch instruction to increase its counter.

3.1.2 Efficient Instrumentation

Clearly we would prefer to have the profiler instrument as little of the code as possible. However, we would also like to obtain the statistics concerning the context of the execution of all the basic blocks. For our purposes, we assume that the profiler cannot be altered, and that it has the capability of returning the frequency information for all the basic blocks in a program. We shall demonstrate how to obtain all the desired statistics, using a given profiler that returns at least the basic block execution frequencies, by instrumenting selected sections of the code before applying the profiler to it.

The reason for the restriction on not altering the profiler is that we would like to consider the profiler as a tool for which the behaviour is limited to that specified by its input/output specifications. We make this clarification because there are techniques for reducing the amount of work done in instrumentation beyond even what we describe here [2]. However, those techniques all involve modifying the profiler. An example application of one of those techniques would be to have the profiler not instrument some nodes whose frequencies could be derived from other nodes' frequencies. While these techniques are useful to those designing profilers, they cannot be applied by end-users of the profiling tool. The techniques we describe show how to use the profiling tool to efficiently return all the information we want without having to know implementation details about the tool.

In maintaining our stance of keeping the compiler and profiler separate, we were forced to modify the program rather than the profiler so that we could obtain the information needed efficiently. Had the implementation been an integrated one, where the compiler and the profiler were built as one tool, then the profiling techniques could have been modified to obtain the information — possibly even more efficiently. Notice that there was, nonetheless, not much loss in efficiency since having the profiler integrated would in no way help the

[1] This is the profiler that was used throughout the development of this thesis work.

compiler to modify the flowgraph. The possible increase in efficiency would have come about because of improvements in profiling techniques which might have avoided doing extra work when the profiler could detect that it was safe to do so.

3.2 Background To The Algorithm

It is clear that it is possible to obtain enough information using just a basic block profiler to deduce all the necessary control flow information since we could simply instrument every branch instruction of the original program (in the manner the profiler would have done it in the first place). But this is inefficient as pointed out earlier and we shall provide upper and lower bounds on the amount of code that will need to be instrumented. In addition, the algorithm described herein, is guaranteed to instrument the minimum amount of code necessary.

To argue formally about our algorithm, we model a program as a control dependency flowgraph where each node represents a basic block from the program. A directed edge connects nodes u and v if it is possible for control to be transferred from basic block u to basic block v. Thus, in the context of flowgraphs, our problem is to deduce all the edge frequencies given a profiler which returns only node frequencies.

A weighted digraph is said to obey Kirchoff's current law (KCL) if for each node in the graph the following three quantities are all equal: the sum of the edge weights leaving the node, the sum of the edge weights entering the node and the node weight.

Note that Kirchoff's current law puts severe constraints on the weighted digraph. In particular, there can be no nodes that do not have edges either going into or coming out of the node. A control dependency flowgraph weighted with frequency counts on the nodes disobeys KCL only at the entry and exit points in the graph, so the law permits analysis of the flowgraph only on the internal nodes (ie those that are neither entry nor exit nodes). This problem can be solved by creating a single entry node going to all other entry nodes and similarly, a single exit node to which all other exits are connected, then finally adding an edge from the exit node to the entry node. We use this construction in our later proofs, but since that last edge added is never instrumented, in practice we do not insert it. The following theorem demonstrates that it is indeed possible to gather all the edge weights using only node weights. It is merely a formal restatement of the informal argument presented above about instrumenting every edge.

Theorem 3.1 *Given a profiler, P which only returns node counts of a program's control dependency flowgraph, it is possible to deduce the edge counts using only the information from P.*

Proof: Let G be the control dependency flowgraph. Then $G = \langle V, E \rangle$ where V is the set of vertices and E the set of edges, represented as ordered pairs so $E \subseteq V \times V$. Let $p(v)$ represent the weight of a vertex, v, in V as assigned by profiler P.

To obtain weights for each edge in E, we construct a graph $G' = \langle V', E' \rangle$ from G by replacing each edge (u, v) with a new node w and new edges (u, w) and (w, v) as in Figure 3.1. This construction is equivalent to instrumenting each edge.

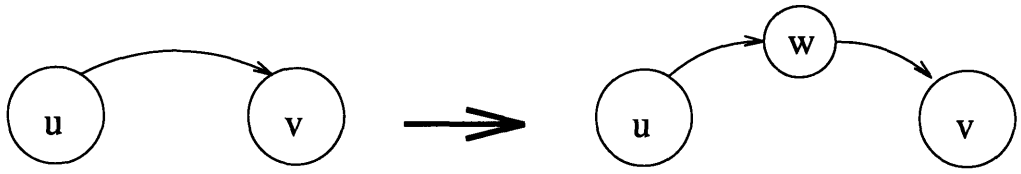

Figure 3.1: Replacing an edge with an instrumenting node

The new graph G' will have the following properties:

1. $|V'| = |V| + |E|$
2. $|E'| = 2|E|$
3. $V \subset V'$
4. $\forall (u,v) \in E \; \exists w \in V' - V$ such that:
   (a) $\{(u,w),(w,v)\} \subset E'$
   (b) $(u,v) \notin E'$
   (c) $\forall x \in V'[((x,w) \in E' \leftrightarrow x = u) \wedge ((w,x) \in E' \leftrightarrow x = v)]$
   (d) The conditions under which edge (u,w) is traversed in G' are exactly the same as for when the edge (u,v) would have been taken in G.

From property 4, $|V' - V| \geq |E|$. So properties 1 and 3 together imply that $|V' - V| = |E|$. Therefore, $\forall v \in V' - V \; \exists$ a unique $(u_1, u_2) \in E$ such that $(u_1, v) \in E'$ and $(v, u_2) \in E'$. Now by KCL, $p(v) =$ weight $(u_1, v) =$ weight (v, u_2). All the conditions of property 4 ensure that weight $(u_1, u_2) =$ weight $(u_1, v) =$ weight (v, u_2). Hence, $p(v) =$ weight (u_1, u_2).

Since the last property guarantees that there is a corresponding node in $V' - V$ for each edge in E, then the information returned by P on G' is sufficient to determine all the edge weights of G. ■

Theorem 3.1 used a construction which involved adding $|E|$ nodes to the graph, where $|E|$ is the number of edges in the original graph. We shall show that the edge information can always be obtained with fewer than $|E|$ nodes added. We shall also show that the algorithm presented is optimal in that it adds the minimal number of nodes required to deduce all the edge weights.

3.3 Description of The Algorithm

The overall algorithm is broken into three stages:

1. Choose the edges to be instrumented

2. Instrument the chosen edges

3. Measure the node frequencies (weights) and solve for the edges.

The most important stage is the first since the edges marked *chosen* will be those to be profiled. The number of these so marked edges will be the determining factor on the time that profiling takes. When we say that this algorithm is optimal, we mean that it chooses the fewest possible edges to allow the rest of edges to be deduced from only the node weights measured.

Before discussing the algorithm in detail, we shall need the following definition.

Definition 3.1 *A weighted adjacency matrix, W, is the matrix constructed from a weighted digraph $G = \langle V, E \rangle$ such that:*

$$W_{ij} = \begin{cases} \textit{weight of edge } (i,j) & \textit{if } (i,j) \in E \\ 0 & \textit{otherwise} \end{cases}$$

The following construction will also be useful in proving the correctness and optimality of the algorithm.

Let A_W be the weighted adjacency matrix of G, a connected graph obeying KCL, and W_V be the column vector of node weights such that W_{V_i} = weight of node i. Using KCL, we can get two systems of equations: one for the outgoing edges of each node, and the other for the incoming edges of each node. That is, if $\mathbf{c}$ is a constant column vector of size $|V|$ such that $\forall i\ c_i = 1$ then we may write:

$$A_W \mathbf{c} = W_V \tag{3.1}$$

$$(\mathbf{c}^T A_W)^T = W_V \tag{3.2}$$

The second equation can be rewritten as:

$$A_W^T \mathbf{c} = W_V \tag{3.3}$$

As an example, consider the graph in Figure 3.2. The nodes and edges have been numbered in some arbitrary order, and the resulting adjacency matrix would be:

$$A_W = \begin{bmatrix} 0 & 0 & e_1 & e_2 \\ 0 & 0 & e_3 & e_4 \\ e_5 & 0 & 0 & 0 \\ 0 & e_6 & 0 & 0 \end{bmatrix}$$

Note that in general, the matrix A_W has no zero rows or columns since every node is connected to at least one other node, and every node has at least one node connected to it. (This was guaranteed by construction and the conditions on G). Therefore, Equations 3.1 and 3.3 represent two systems each with $|V|$ equations and involving the same variables.

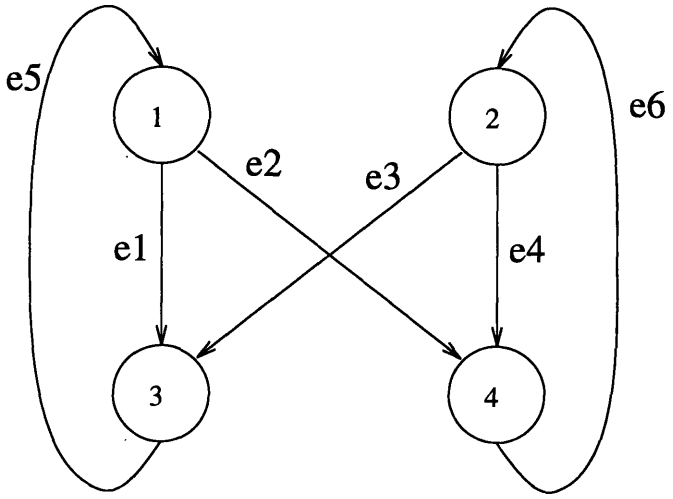

Figure 3.2: An example of a control flowgraph

3.3.1 Choosing the Edges to be Instrumented

First we construct the weighted adjacency matrix, A_W, for the given control flowgraph, G, using a unique variable to represent each edge weight, since they are as yet unknown. There are two possible marks that can be assigned to a variable: *known* or *chosen*, and at the beginning all variables in A_W are unmarked. We proceed as follows:

1. For each row, A_{W_i}, of A_W,

   If A_{W_i} contains exactly one unmarked variable then mark that variable as *known*.

2. For each column, $A^T_{W_i}$, of A_W,

   If $A^T_{W_i}$ contains exactly one unmarked variable then mark that variable as *known*.

3. If all variables have been marked (either *known* or *chosen*) then halt.

4. If at least one variable was marked in the most recent pass then go back to step 1.

5. Choose any variable that is unmarked and mark it *chosen*.

6. Go back to step 1.

3.3.2 Instrumenting the Chosen Edges

After all the variables have been marked, we instrument the edges corresponding to the variables that were marked *chosen*. In the model of the graph, this means adding a node with exactly one incoming and one outgoing edge. The source of the incoming edge and the destination of the outgoing edge are exactly the same as the source and destination, respectively, of the edge being instrumented. The edge being instrumented is then deleted

from the graph. In terms of the program, this means that the branch instruction corresponding to the edge being instrumented is modified so that its target is now the address of the instrumenting code. At the end of that instrumenting code is a branch instruction with the same branch target as that of the original branch instruction.

Under this construction, all the conditions of property 4 in the above theorem are satisfied. In particular, condition (d) is satisfied since only the branch target of the branch instruction is modified. Hence, any condition which would have transferred control to the original branch target address will now transfer control to the basic block containing the instrumentation code. Under the graph model this means that whenever the removed edge would have been traversed, the incoming edge of the new node will be traversed.

3.3.3 Computing the Results

In the final stage of the algorithm, we pass the modified program to the profiler which returns the execution frequencies of all the basic blocks in the modified program (ie the node weights). Note that the order in which the edge variables were marked *known* implies the order in which they should be solved. Since any variable that becomes the lone variable in its column or row after another was marked *known* will only be deducible after that marked variable has been solved for.

As an optimization, we constructed the expression for each variable, as it was being marked, in terms of the node frequencies and the other edge frequencies. Then after the profiler returned the frequencies for all the nodes and the instrumented edges, the values were substituted into the expressions which were then solved in the order that they were constructed (the same as the order in which the variables were marked). This optimization allowed the all the edge frequencies to be deduced in one pass, since no expression would be attempted to be evaluated until all the information for it was available.

3.4 Correctness of the Algorithm

We first show that the algorithm is correct in that it will return the correct execution frequencies for the edges so long as all the reported node frequencies are correct.

Claim 3.4.1 *The algorithm for labelling the variables terminates.*

Proof: Step 4 guarantees that on each pass, at least one variable will get marked either *known* or *chosen*. Since variables are never remarked then the total number of unmarked variables decreases by at least 1 per pass. Since the total number of unmarked variables at the beginning of the stage is finite then the condition in Step 3 will eventually evaluate to true, causing the algorithm to halt. ■

Claim 3.4.2 *If a variable e is marked* chosen *then given the node weights of the flowgraph, the value of e can be uniquely determined.*

Proof: By construction, the frequency of the instrumenting node for the edge corresponding to e is equal to e. Since this frequency will be returned by the profiler after stage 3, the value of e will be uniquely determined. ■

Claim 3.4.3 *If a variable e is marked* known *in some pass t, then given the node weights of the flowgraph and the values of all the variables already marked on pass t, the value of e can be uniquely determined.*

Proof: We prove this by induction. If a variable is marked *known* on the first pass then it must have been the only variable in either its row or column according to steps 1 and 2. The value of this variable is immediate once the frequencies have been measured as implied by equation 3.1 and equation 3.3.

Assume now that on pass $t-1$ every variable that was marked *known* was correctly computable. If e is marked on the t^{th} pass then it must have been the only unmarked variable in either its row or its column on pass $t-1$. Since e will be marked on pass t, it must be that all the other variables in its row (column) have been marked either *known* or *chosen*. Given the values of all the variables marked before pass t, we can find the sum of all the other variables in the same row (column) as e. The value of e is then given by the difference of the measured node weight and the computed sum of previously marked variables (i.e. the edge weights) as implied by equation 3.1 (equation 3.3). ■

It follows from the preceding claims that the algorithm is correct. The first claim guarantees that all the variables get marked and the second and third show that a marked variable can be correctly computed. Hence all variables will be correctly computed. That is to say that the algorithm returns correct edge weights based upon the node weights returned by the profiler.

3.5 Optimality of The Algorithm

The proof of optimality will require the use of a few Theorems from Linear Algebra.

Theorem 3.2 *If A is an $m \times n$ matrix with rank n, and $\mathbf{x}$ is a $n \times 1$ vector, then the equation: $A\mathbf{x} = \mathbf{0}$ has $\mathbf{x} = \mathbf{0}$ as its only solution, if indeed, that solution exists.*

Corollary 3.2.1 *The equation $A\mathbf{x} = \mathbf{b}$, for some $m \times 1$ vector $\mathbf{b}$ has a unique solution.*

Theorem 3.3 *Exchanging any two columns of the coefficient matrix of a system of equations will not alter the constraints of the system on the variables.*

Proof: Consider the system

$$A\mathbf{x} = \mathbf{b}$$

If A' is obtained from A by exchanging columns i and j, then we can construct $\mathbf{x}'$ and $\mathbf{b}'$ by exchanging elements i and j of $\mathbf{x}$ and $\mathbf{b}$ respectively. Now, the system $A'\mathbf{x}' = \mathbf{b}'$ is identical to the given system, hence the constraints of the two systems are also identical. ■

3.5.1 An Alternate Matrix Representation

Recall equations 3.1 and 3.3, within A_W there are exactly $|E|$ variables each one representing an edge in the graph. So we could rewrite each system in terms of a coefficient matrix of 1's and 0's, the vector of all the $|E|$ variables, and the $|V|$ node weights. That is, for equation 3.1, say, we could write:

$$M_R W_E = W_V$$

where M_R is the coefficient matrix and W_E is the vector of edge weights. Note that M_R is a $|V| \times |E|$ matrix and that by writing the right hand side as W_V, we have forced the ith row of M_R to correspond in some manner to the ith row of A_W and hence to the ith node. This correspondence is actually quite straightforward to see, we simply have to make the expression $M_{Ri}W_E$ algebraically equivalent to the expression $A_{Wi}\mathbf{c}$, where M_{Ri} and A_{Wi} refer to the ith rows of M_R and A_W respectively. So for example, earlier we had

$$A_W = \begin{bmatrix} 0 & 0 & e_1 & e_2 \\ 0 & 0 & e_3 & e_4 \\ e_5 & 0 & 0 & 0 \\ 0 & e_6 & 0 & 0 \end{bmatrix}$$

From the first row of A_W, we see that variables e_1 and e_2 sum to the frequency of node 1 so that M_R will have two 1's in the first row in the column positions corresponding to e_1 and e_2. The rest of M_R is given below.

$$M_R = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

This construction yields the following properties within M_R:

1. M_R is a $|V| \times |E|$ matrix of 1's and 0's.
2. Each row of M_R contains at least one 1.
3. Each row of M_R has at most $|V|$ 1's in it.
4. Each column of M_R has *exactly* one 1 in it.

The first property is obvious from the dimensions of W_E and W_V. The second and third properties come from the fact that none of the rows of A_W were all zero and that its dimensions were $|V| \times |V|$, which means that there was at least one edge weight variable but at most $|V|$ of them in a row. The last property may seem a little startling at first, but is actually quite simple to see why. By the constraint governing the construction of M_R we see that if element M_{Rij} is a 1, then the variable W_{Ej} appears in the row A_{Wi}. Now since a variable can be in at most one row of A_W (that is, every edge has at most one source node) then each column of M_R has at most one 1. Also since the variable was obtained from A_W in the first place, it must sit in some row. (That is every edge has at least one source node) Therefore, each column of M_R has at least one 1; hence exactly one 1. The last property is, in fact, just another way of stating that every edge has exactly one source node.

We can do a similar restatement of the system implied by Equation 3.3 to yield a coefficient matrix M_C. The important difference is that the constraint on the construction of M_C is that $M_{C_i} W_E$ must now be algebraically equivalent to $A^T_{W_i}\mathbf{c}$. Now, M_C has all the same properties as M_R does, but they are true because of the column properties of A_W instead of its row properties (since the rows of A^T_W are the columns of A_W). The equivalent restatement of property 4 for M_C would be that every edge has exactly one destination node. So the corresponding M_C for our current example would be

$$M_C = \begin{bmatrix} 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \end{bmatrix}$$

Finally, we can combine the two systems into one to write:

$$\begin{bmatrix} M_R \\ M_C \end{bmatrix} W_E = \begin{bmatrix} W_V \\ W_V \end{bmatrix} \tag{3.4}$$

or

$$MW_E = W_N \tag{3.5}$$

where $M = \begin{bmatrix} M_R \\ M_C \end{bmatrix}$ and $W_N = \begin{bmatrix} W_V \\ W_V \end{bmatrix}$

That is, M is the $2|V| \times |E|$ matrix obtained by placing the rows of M_R above those of M_C, and W_N is the $2|V| \times 1$ column vector obtained by concatenating W_V to itself. This final form will allow us to prove the theorems following.

3.5.2 Bounds for the Number of Instrumented Edges

Theorem 3.4 *Given a control dependency flowgraph $G = \langle V, E \rangle$ with no unreachable nodes, if $|E| \geq 2|V|$ then the edge weights cannot, in general, be uniquely determined if given only the node weights of the unmodified graph, G.*

Proof: We shall first construct a graph $G' = \langle V', E' \rangle$ which has G as a subset, and will obey KCL at every node. To do this, if there is at least one exit node, we create a single entry node and a single exit node which are linked by an edge going from the exit node to the entry node. (Note that we will always have at least one entry node). Formally, let S and T be the sets of entry and exit nodes respectively. We obtain V' and E' as follows:

$$V' = \begin{cases} V \cup \{s,t\} & \text{if } |T| \geq 1 \\ V \cup \{s\} & \text{otherwise} \end{cases}$$

$$E' = \begin{cases} E \cup (\{s\} \times S) \cup (T \times \{t\}) \cup \{(t,s)\} & \text{if } |T| \geq 1 \\ E \cup (\{s\} \times S) \cup \{(s,s)\} & \text{otherwise} \end{cases}$$

where s and t are distinguished nodes not already in V. Note that there are no nodes in G' without either incoming or outgoing edges.

G' obeys KCL by construction and a consistent set of weights is obtainable if the weights of nodes s and t and edge (t,s) are all set to 1. This is also consistent with the notion that

the program corresponding to the flowgraph exits once. (Indeed it would be unclear as to what it would mean for there to be a weight greater than 1 on node t)

If it were possible to solve for the edge weights of G given the conditions in the theorem statement, then it would also be possible to solve for the edges in G' since all the edges in $E' - E$ are immediately determined once the node weights are known. Hence, by showing that it is impossible to solve uniquely for the edges of G' we can conclude that the theorem statement is true.

Case 1: $|E| > 2|V|$
The two systems above yield $2|V|$ equations in the edge weight variables, therefore in general, there are not enough equations to solve for all edges.

Consider matrix M: there are $2|V|$ rows, therefore $\text{Rank}(M) \leq 2|V|$. Since the number of columns is equal to $|E|$, Theorem 3.2 implies that the solution to the vector $\mathbf{w}$ is not unique. Thus, all the edges cannot be uniquely solved in general. The edges can possibly be solved uniquely if enough of them can be determined to be zero – by being connected to a node with zero weight – so as to reduce the number of unknown edge weights to less than $2|V|$.

Case2: $|E| = 2|V|$
Consider matrices M_R and M_C. In both matrices the sum of the rows is equal to $\mathbf{c}^T$. Therefore, in matrix M at least one row can be reduced to all zeros. Hence $\text{Rank}(M) < 2|V|$ which by Theorem 3.2 implies that the edges cannot be uniquely solved in general. ■

Theorem 3.4 tells us that if the given graph has at least as many edges as twice the number of vertices then the node frequencies alone will not be sufficient to compute the edge frequencies uniquely in the general case. Note that we are not considering the added information that there might be in the actual numbers. For example, if we knew that the frequency at a particular node was zero then all the edges going into and leaving it would be forced to have zero weight. Rather, the theorem refers to the *a priori* ability to determine the edge weights — that is, before we know what any of the node weights actually are, we want to know how many edges we will be guaranteed of being able to solve for.

Having thus established an upper bound on the number of edges that we could hope to being able to solve, we can also establish a lower bound. That is, we can determine what the minimum number of edges is so that we are guaranteed to being able to solve for all their weights uniquely only from the node frequencies. This lower bound is stated and proved in Theorem 3.5.

Theorem 3.5 *Given a control dependency flowgraph $G = \langle V, E \rangle$ with no unreachable nodes, if $|E| \leq |V| + 1$ then the edge weights can, in general, be uniquely determined if given only the node weights of the unmodified graph, G.*

Proof: Case 1: $|E| < |V| + 1$
Since there are no nodes without edges leaving or going into them, then $|E| \geq |V|$. So this case reduces to $|E| = |V|$. In this case, M is a $2|V| \times |E|$ matrix. All the properties of the sub-matrices M_R and M_C, of which M is composed, still hold. So from property 2, we see that each row of M_R and M_C contain exactly one 1. There are exactly $|V|$ different

possible rows of length $|V|$ that contain exactly one 1, therefore every row that appears in M_R must also appear in M_C. Hence by row operations on M, we can obtain exactly $|V|$ rows of zeros. If we choose to reduce those rows from the M_C sub-matrix, we see that

$$Rank(M) = Rank(M_R) = |V|$$

So by Theorem 3.2, there is a unique solution to W_E.

Case 2: $|E| = |V| + 1$
By the pigeon-hole principle, there will be exactly one row, M_{Ri}, in the M_R sub-matrix and exactly one row, M_{Cj}, in the M_C sub-matrix that will contain two 1's. Now, it cannot be the case that M_{Ri} and M_{Cj} contain both their 1's in the same column positions since that would imply that there were two distinct edges going from node i to node j which violates the definition of the graph. Therefore, there is at least one column of M that contains a 1 in row M_{Ri} but not in row M_{Cj}. Since that column must have two 1's, then the 1 from M_C is the only 1 in the row that contains it. Call that row M_{Ck}. Then M_{Ck} and M_{Ri} are two linearly independent rows which involve two edge variables. Therefore, both those variables can be determined given the frequencies of the nodes. A similar argument holds for M_{Cj}, so both the variables that it involves can be deduced only from the node frequencies. Since all the other rows contain only one 1 each, the values of the edge variables corresponding to the positions of those 1's are immediate from the node frequencies. Hence all the edge variables can be deduced. ■

3.5.3 Precisely Expressing the Required Number of Instrumented Edges

We now have a upper bound on the number of edges to tell us when we can solve uniquely for them all, and a lower bound to tell us when we certainly cannot solve for the edge frequencies uniquely. But what about when the number of edges is between these two bounds? That is, we would like to know what criterion there is – if any – that determines whether or not we can solve the edges uniquely given that $|V|+1 < |E| < 2|V|$. It turns out that it is when $|E|$ is in this range that the number of edges that need to be instrumented is determined only by the topology of the graph. This is true because the topology of the graph determines the dependencies among the rows and columns of M and hence its rank and the sizes of its null spaces. The rank of M, in turn, determines the number of edges that can be determined uniquely, and so by observing the rank of M, we can decide how many edges will have to be instrumented. We formalize the preceding argument, and produce the exact number of edges that will be instrumented in terms of the row and column space attributes of M.

Lemma 3.1 *The minimum number of edges that need to be instrumented in order for the rest of them to be determined uniquely is* $|E|-Rank(M)$.

Proof: Instrumenting an edge forces the profiler to measure its frequency directly, so is equivalent to replacing the variable corresponding to the edge in the vector W_E with the measured frequency. Let us suppose that we instrumented n edges and we were able to solve for the rest uniquely. Then we could replace n of the variables in W_E with constants representing the frequencies measured. By Theorem 3.3, we can exchange the elements of

W_E and the corresponding columns of M in such a way as to make the last n entries in W_E be all constants. Thus, the elements of W_E are partitioned into two sub-vectors: a $(|E|-n)$-length vector, W'_E, containing all variables, and a n-length vector, W''_E containing all constants. Correspondingly, M is partitioned into two sub-matrices: a $2|V| \times (|E|-n)$ matrix, M', and a $2|V| \times n$ matrix, M''. So Equation 3.5.1 becomes:

$$\begin{aligned} MW_E &= [M'|M''] \begin{bmatrix} W'_E \\ W''_E \end{bmatrix} \\ &= M'W'_E + M''W''_E = W_N \\ \text{so,} M'W'_E &= W_N - M''W''_E \end{aligned} \tag{3.6}$$

The right hand side of the last equality will be a known vector after the profiler returns the node weights, and since by assumption we could solve the original system after instrumentation, Theorem 3.3 guarantees that we can solve the derived system. Then, since M' has $|E|-n$ rows, Theorem 3.2 implies that $\text{Rank}(M') = |E|-n$. But by construction, $\text{Rank}(M') \leq \text{Rank}(M)$ so $|E|-n \leq \text{Rank}(M)$. That is, $n \geq |E| - \text{Rank}(M)$. ■

Corollary 3.5.1 *The number of edges that need to be instrumented is $dim(\mathcal{N}_R(M))$ where $\mathcal{N}_R(M)$ is the right null space of M.*

Proof: Since M is a $2|V| \times |E|$ matrix,

$$Rank(M) + dim(\mathcal{N}_R(M)) = |E|$$

If n is the minimum number of edges that need to be instrumented, Lemma 3.1 shows that

$$\begin{aligned} n &= |E| - Rank(M) \\ &= dim(\mathcal{N}_R(M)) \end{aligned} \tag{3.7}$$

$$\tag{3.8}$$

■

Corollary 3.5.2 *Provided $|E| \geq |V|$, $|V| \leq Rank(M) \leq 2|V|-1$*

Proof:

1. Theorem 3.4 tells us that if $|E| \geq 2|V|$ then we need at least one instrumented edge. In particular, if $|E| = 2|V|$ then $n = |E| - \text{Rank}(M) \geq 1$, which implies that $\text{Rank}(M) \leq 2|V|-1$.

2. Theorem 3.5 indicates that we don't need any instrumented edges if $|E| \leq |V|+1$. Since there are no unreachable nodes in the flowgraph and it obeys KCL, we also know that $|E| \geq |V|$. So for $n = |E| - \text{Rank}(M) = 0$, we obtain $\text{Rank}(M) = |E|$ when $|V| \leq |E| \leq |V|+1$ implying that $\text{Rank}(M) \geq |V|$ so long as $|E| \leq |V|+1$. Now we note that $\text{Rank}(M)$ must be monotonically nondecreasing since when new edges are added, at worst, they are all in the null space of the matrix, M, which does not change the value of $\text{Rank}(M)$. Hence, we have $\text{Rank}(M) \geq |V|$ in the general case when $|E| \geq |V|$.

■

Corollary 3.5.3 puts bounds on the rank of the edge coefficient matrix, M, given that there are enough edges to connect all the vertices to obey KCL. The exact value of Rank(M) is dependent on the topology of some subset of the graph containing all the vertices and at most $2|V| - 1$ edges, and not on the total number of edges.

3.5.4 Proof of Optimality

We shall now show that the algorithm will select exactly $\dim(\mathcal{N}_R(M))$ edges to be instrumented and hence that it is optimal. First we show that by obtaining the value of a variable through instrumentation, we can reformulate the system of equations specified by Equation 3.5.1 using a matrix M' with one less column and without using the variable that was marked "chosen" . We then prove that the conditions under which a variable is marked "chosen" guarantee that none of the variables unknown at the time could have been derived from the ones already marked. That is, the variable marked "chosen" is independent of the other edge variables.

These two conditions imply that if M' is the coefficient matrix obtained after reformulating the system and M was the matrix before, then $\dim(\mathcal{N}_R(M'))$ =$\dim(\mathcal{N}_R(M)) - 1$ since only one variable was marked and it was an independent variable. Since each instrumentation reduces the dimension of the right null space of the coefficient matrix by 1, we can instrument at most $\dim(\mathcal{N}_R(M))$ edges. By Lemma 3.1 we know we cannot instrument any less than $\dim(\mathcal{N}_R(M))$ edges implying that we instrument exactly $\dim(\mathcal{N}_R(M))$ edges.

Independence of *chosen* variables

For this part of the proof of optimality, we use matroids as a repesentation of matrices, to simplify the reasoning about independent rows. The section on matroids describes matroids formally and shows how the set of rows and the sets of linearly independent rows of a matrix form a matroid. In this instance, we define a matroid $\mathcal{M} = (\mathcal{S}, \mathcal{I})$ where $\mathcal{S}$ is the set of rows of M, and a subset of $\mathcal{S}$ is in $\mathcal{I}$ if the rows it contains are all linearly independent. As constructs for the proof, we will also need the followig functions on vectors and sets of vectors. For $\mathbf{r} = \langle r_1, r_2, \ldots, r_n \rangle$, an n-dimensional vector we can define:

$$C_v(\mathbf{r}) = \{i | r_i \neq 0\}$$

and extending this to sets of vectors, we get

$$C(A) = \bigcup_{\mathbf{r} \in A} C_v(\mathbf{r})$$

Lemma 3.2 *If every row of M contains at least two 1's then $|C(A)| > |A|$ for every set A in $\mathcal{I}$*

Proof: Let A be some subset of $\mathcal{S}$ in $\mathcal{I}$. We prove $|C(A)| > |A|$ by induction on the subset sizes of A. When $|B|$=1 for some $B \subset A$ then $|C(B)| \geq 2$, so the lemma is true for the

basis case of a single row. Assuming that the lemma holds true for all the proper subsets B of A of size k, by the exchange property of M we know there is some $\mathbf{r} \in A - B$ such that $B \cup \{\mathbf{r}\} \in \mathcal{I}$; that is, $\mathbf{r}$ is independent of all the rows in B.

Let $B' = B \cup \mathbf{r}$, we need to show that the inductive hypothesis implies that $|C(B')| > |B'| = |B| + 1$. Since $|C|$ is monotonic increasing we know $|C(B')| \geq |C(B)|$ and $|C(B)| \geq |B| + 1$ by hypothesis. So $|C(B')| \geq |B'|$ and we only need to show that it is impossible for $|C(B')| = |B'|$.

Suppose $|C(B')| = |B'|$ and consider t, the total number of 1's in all the vectors within B', we can obtain the possible range of values for t from the assumption and the properties of M. Since there are at least two 1's per row and at most two 1's per column, we obtain $2|B'| \leq t \leq 2|C(B')|$. But $|C(B')| = |B'|$ by supposition, therefore $t = 2|C(B')| = 2|B'|$. This means that every row in B' contains exactly two 1's and for each 1 in each row, $\mathbf{r}$, of B', there is another row $\mathbf{r}'$ in B' that contains a 1 in the same index position. From the construction of M, one of $\mathbf{r}$ and $\mathbf{r}'$ originated from M_R and the other from M_C.

Hence we conclude that $|B'|$ is even, and that the sum of all the rows in B' that came from M_R is a vector with a 1 in every index position in $C(B')$. The same is true for all the rows of B' that came from M_C, and by subtracting these two sums, we obtain a zero vector implying that there is a linear dependence within the vectors of B'. This contradicts the exchange property of $\mathcal{M}$ from which B' was constructed, and so it must not be the case that $|C(B')| = |B'|$. That is $|C(B')| > |B'|$ follows from the inductive hypothesis, and so we conclude that for every A in $\mathcal{I}$, $|C(A)| > |A|$. ■

Theorem 3.6 *If the algorithm marks a variable "chosen" then none of the unmarked edge variables immediately prior to that marking could have been determined solely from the flow constraints.*

Proof: First we note that the algorithm only marks an edge variable "chosen" when every column or row within the current weighted adjacency matrix contains at least two variables. The corresponding state of M is that every row of M contains at least 2 1's. Thus the stipulation of Lemma 3.5.4 is satisfied.

To be able to determine the value of an edge variable, there has to be a set of independent rows of M that do not involve any more variables than there are rows in the set. Formally, there must be some A in $\mathcal{I}$ for which $|C(A)| = |A|$ since $|C(A)|$ is the number of variables that are involved with a system containing a matrix made up of the rows within A. Therefore, invoking Lemma 3.5.4, we see that it is never the case for any A in $\mathcal{I}$ that $|C(A)| = |A|$ and so we conclude that indeed there is no set of rows of M which could lead to a unique solution for the variables still unknown. ■

Reformulation After Instrumentation

Since marking a variable "chosen" means that its corresponding edge will be instrumented, the value of the variable can be considered as given. Let the selected edge be edge variable i, then we can reformulate the system of equations by:

1. Deleting column i from M to obtain M'.
2. Deleting element W_{E_i} from the vector W_E to get $W_E{}'$.
3. If the two rows of M that contain a 1 in the i-th column are the j-th and k-th rows then we obtain $W_N{}'$ from W_N by assigning $W_{N_j}{}' = W_{N_j} - W_{E_i}$ and $W_{N_k}{}' = W_{N_k} - W_{E_i}$

For example,

$$\begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 \end{bmatrix} \begin{bmatrix} e_1 \\ e_2 \\ e_3 \\ e_4 \\ e_5 \\ e_6 \\ e_7 \end{bmatrix} = \begin{bmatrix} n_1 \\ n_2 \\ n_3 \\ n_1 \\ n_2 \\ n_3 \end{bmatrix}$$

would be transformed as follows if edge 5 were chosen to be instrumented:

$$\begin{bmatrix} 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} e_1 \\ e_2 \\ e_3 \\ e_4 \\ e_6 \\ e_7 \end{bmatrix} = \begin{bmatrix} n_1 \\ n_2 - e_5 \\ n_3 \\ n_1 \\ n_2 \\ n_3 - e_5 \end{bmatrix}$$

Note that the new matrix M', like M also has following properties:

1. All the entries are either 0 or 1.
2. Each row has at most $|V|$ 1's in it.
3. Each column has *exactly* two 1's in it.

These properties, invariant on the coefficient matrix as the algorithm proceeds, guarantee that at every choice of a variable for instrumentation, all the conditions for Lemma 3.5.4 are satisfied. Therefore Theorem 3.6 holds for each successive M', and hence we are guaranteed that every instrumentation was necessary.

3.6 Running Time of the algorithm

Instrumenting the chosen edges and solving the expressions for the edge variables are relatively inexpensive steps especially since they can both be done in one pass of the variables. Therefore, the running time of the whole algorithm, not including the time for the profiler to return the frequencies, is dominated by the time to choose the edges to instrument.

The algorithm for choosing edges to be instrumented can search through at most $|V|$ rows and columns while checking for lone variables. Each check is cheap since there can be an array for each row and column which keeps track of the number of variables in that row or

column and which variables are there. When a variable is marked, it is removed from the list for its row and its column and the sizes of the lists updated. This was, in fact, how the algorithm was actually implemented. Since all this work is done for each variable that is marked and we mark all the variables, then the total number of steps is $O(VE)$.

The running time of the algorithm is actually irrelevant for the purposes of this context, but was included for the sake of completeness. The time that the algorithm takes will be reflected in how long the compiler takes to complete a compilation, but will not affect the profiling time at all. Since it is the latter that we are mostly concerned with when using this algorithm, we try to minimize the number of edges instrumented.

3.7 Matroids

A matroid is an algebraic structure consisting of a pair of sets $(\mathcal{S}, \mathcal{I})$ which has the following properties:

1. $\mathcal{S}$ is finite.
2. $\mathcal{I}$ is a non-empty set of subsets of $\mathcal{S}$ such that if $A \in \mathcal{S}$ and $B \subseteq A$ then $B \in \mathcal{I}$. This property is called the *hereditary* property of $\mathcal{I}$.
3. $\mathcal{I}$ satisfies the *exchange* property: if A and B are both in $\mathcal{I}$ and if $|A| > |B|$ then there is an element $x \in A - B$ such that $B \cup \{x\} \in \mathcal{I}$.

We can obtain a matroid from a matrix by using the rows of the matrix as $\mathcal{S}$, the finite set of the matroid pair, and considering a subset of it independent (i.e. belonging to $\mathcal{I}$) if the rows in that subset are linearly independent. Using the columns instead of the rows also yields a matroid, this type of matroid, derived from a matrix, is called a matric matroid [7].

Although originally inspired by their application to matrices [19], matroids are applicable to many other areas of Computer Science. For example, the graphic matroid, derived from a graph where the set $\mathcal{S}$ is the set of edges and a subset, A, of $\mathcal{S}$ is considered to be independent (in $\mathcal{I}$) if and only if A is acyclic. In particular, matroids are very useful for determining when a greedy algorithm yields an optimal solution, because they exhibit the optimal-substructure property. This property states that the optimal solution to a problem contains optimal solutions to its subproblems, so that local optimizations at each stage yield a globally optimal solution. The minimum-spanning-tree problem can be expressed and solved using matroids, as well as a variety of other algorithms that can be solved greedily. Given that the algorithm described in this chapter is a greedy one, it is no wonder that matroids were applicable.

3.8 Summary of Results

It has been shown that edge weight profiling can be done efficiently using only a node weight profiling tool. The number of edges chosen to be instrumented are minimized and the edges so chosen are not affected by the topology of the graph. The topology of the

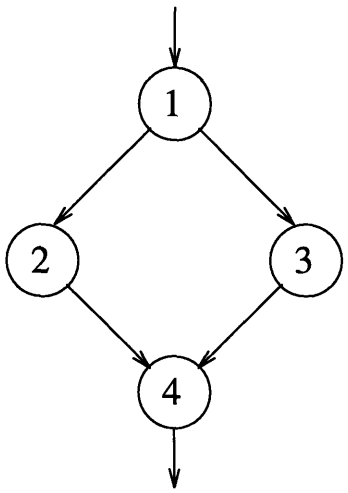

Figure 3.3: Redundant Node Information

graph yields particular properties of the null spaces of the corresponding matrix for the flowgraph, which affect the number of edges that are chosen for instrumentation, but not which ones in particular.

3.9 Discussion

The algorithm is optimal in the number of edges that it instruments, and works on arbitrary flowgraphs. We are guaranteed that it will never instrument all the edges and so in all instances, the algorithm performs better than a naive solution to the problem. It processes the graph in a purely syntactic fashion in that its only concern is the topology of the graph and not any meaning that may be ascribed to the nodes and edges. For example, one possible optimization would have been to check if loop bounds were constant. If so, then the frequencies for some of the edges within the subgraph representing the loop might be deducible. In particular, the frequency of the back edge for the loop would be known. However, this example is a semantic optimization in that it looks at the content of the node and tries to deduce properties of the graph based on the meaning of that content.

The constraints imposed by the compiler and profiler being separate prevent even some syntactic optimizations. For example, if we had the situation in Figure 3.3 then we would not need the frequency information for node 4, since it would be equal to that of node 1. However, such optimizations could not be implemented, because the profiler is assumed to return all the basic block information without making any distinctions between them. Of course, that particular optimization could very well be one within the profiler itself, but the point is that the compiler has no control over how the nodes are measured, so it cannot attempt to improve the performance in general by being clever about the nodes.

Another possible optimization that was not explored is the use of the values obtained in previous runs to hint at which variables to choose to instrument. First of all, if a node

had frequency zero then automatically, all the incoming and outgoing edges would have frequency zero. Secondly, when we have to choose an edge to be instrumented, instead of choosing any available edge, we may actually wish to bias our choice towards the edge that tends to be traversed less frequently. These optimizations involve feedback not only within the compilation process, but also in the profiling. Therefore, they are quite practical in an adaptive dynamic setting where programs are constantly being profiled, possible in the static setting where a database of application profiles is kept, but infeasible in the setting of this implementation. Were we to attempt to use feedback in the profiling phase of the compilation, we would have to do two passes at the profiling, which is already the slowest step by orders of magnitude, in the whole compilation process.

3.10 Bibliographic Notes

The *xprof* profiler was implemented by Ravi Nair at IBM T. J. Watson Research Center [14]. This profiler had the feature that it could profile binaries that had not been instrumented by the compiler, unlike other profiling programs like prof and gprof. He was also instrumental in the development of the algorithm presented here.

Ball and Larus present methods for optimally profiling programs [2] when the profiler has complete control over where counters ought to be placed in the flowgraph to obtain all the information accurately. In our implementation, we are constrained to instrument only edges and every node will be profiled regardless of whether it needs to be or not. Therefore, the constraints on this sytem are slightly more restrictive than those under which they worked. Some of their lower bounds and ideas concur exactly with our results, despite this disparity of constraints.

Cormen, Leiserson and Rivest present a more detailed description of matroids than was presented here [7]. The section on matroids was taken almost wholly from the book. They also indicate sources for a generalization on matroids called greedoids.

Chapter 4

Optimizing with Feedback

4.1 Introduction

The optimizations that can take advantage of feedback information are probably the most interesting part of the whole compilation process; they certainly are among the least understood aspects of it. There are two related but distinct issues that are important and need to be considered:

1. What kind of data should be used in the feedback process to drive the optimizations.
2. How should feedback optimizations be ordered, among each other and among conventional optimizations.

One thing to bear in mind is that feedback based optimizations are probabilistic. That is, they are not guaranteed to improve the performance of the code fragment that was transformed under all inputs. Unlike traditional optimizations that are, for the most part, oblivious of the input to the program, and will generally reduce the execution time for the code fragment that was transformed, feedback based optimizations are highly input dependent. This means that it is possible for a feeback based transformation to actually degrade the performance of a code fragment on certain classes of inputs. The justification, though, for using this class of transformations is that based on the feedback information, we expect that if the future usage patterns of the program correlate well to the past usage patterns, then the transformations we perform will have a net benefit for most of those anticipated inputs.

The business of assessing an optimization that is based on feedback data is an open topic. Indeed, what the right measurement for assessing the benefit of an optimization is, has many possible answers, no one of which is easily refuted. We provide a more detailed discussion of this in Chapter 5.

In the remainder of the chapter, we shall discuss the optimizations that could have been done and how they would have been implemented using the features provided by the system. We shall keep the discussion in the light of the two issues raised earlier, and how their consideration would affect the implementation of the particular optimization under question.

4.2 Trace Based Optimizations

Trace based optimizations are those that are based on traces constructed within the control flowgraph using the frequency information of the feedback system. Fisher describes the process of Trace Scheduling as a technique for microcode compaction on VLIW machines [10]. The process involves grouping instructions that are likely to be executed sequentially according to the profiled data. These instructions are collectively known as a trace and are usually larger than a basic block; Fisher's technique constructs these traces allowing more parallelism to be extracted so that the wider instructions result in shorter code. To maintain correctness, extra code has to be included to ensure that when control leaves a trace, the state of the program is as it would have been before the trace was constructed. Ellis details the algorithms involved in building traces from basic blocks and describes the techniques for maintaining correctness and controlling code explosion [9]. Akin to trace scheduling is *trace copying* described by Hwu and Mahlke [20]. Trace copying is very similar to trace scheduling, but has simplified maintenance routines with the cost of higher code expansion than trace scheduling. All the optimizations that pertain to traces can be implemented with traces constructed by either method.

Although originally designed for VLIW machines, trace scheduling and its related techniques is applicable to uni-processor machines because the traces, typically larger than basic blocks, expose more opportunities for code optimizations. Basic blocks are typically no more than five instructions long, and so peephole optimizations on these basic blocks are limited in scope and effect. The larger traces allow classic optimizations to exploit code properties that exist across basic blocks. Hence, trace constructing techniques coupled with classic code optimizations can provide very effective methods for improving code performance, as described by Chang, Mahlke and Hwu [4].

These optimizations should probably be attempted before and probably even em instead of the classic code optimizations. For transformations, like copy propagation, constant folding and common subexpression elimination, that do not modify the structure of the flowgraph, the global trace based optimizations would simply replace the classic ones since the basic blocks would only be exposing a subset of the code exposed by traces. Other types of optimizations that could modify the topology of the graph might present some difficulty in maintaining the probabilities assigned to the edges in the graph, and so their feedback based counterparts ought to be done first, and then, if necessary, another pass of the traditional optimizations could be performed.

Although these optimizations were not implemented, this was no limitation of the design of the feedback mechanism. The primary type of feedback information would be the edge frequencies of the flowgraph which were already available in the system. These frequencies would be used essentially for trace construction since the optimizations themselves are mostly unchanged and are simply applied to the larger traces instead of to the basic blocks. Therefore, from the point of view of having the feedback information, the system proves to be adequate for implementing at least this class of optimizations.

4.3 Non-Trace-Based Optimizations

This class of optimizations use the feedback information directly on the code from the flowgraph and may use more than just the edge frequencies. The type of information being fed back is of particular importance here because it determines what aspects of the program may be improved. For example, if the feedback information contains cache-miss information per basic block then a possible optimization might be to rearrange the layout of the code purely for the sake of reducing the memory access latencies due to cache misses. As another example, suppose the feedback contained statistics about the contents of variables on a per basic block basis. We could imagine that if a dispatch loop, for example, were being optimized, we may be able to notice correlations with the procedures called and the data observed. From this, the compiler might decide that it is beneficial to inline a procedure, or to reorder code layout to improve instruction cache hit ratios. The point of both those examples was to show that when we consider more information than simply the execution frequencies, we provide opportunities for improving performance in ways that are beyond the typical considerations for optimization.

4.3.1 Using Execution Frequencies

An important optimization that would benefit immensely from feedback execution frequencies is branch prediction. The idea is to exploit the unequal delays along the two branch paths (in the RS6000 in particular) by rearranging the code. The reason for the unequal branch delays is that the RS6000 assumes that the fall-through branch is taken whenever a conditional branch instruction is encountered. Therefore, the taken branch path has a longer delay than the fall-through branch path and so conditional branches are better off arranging the tested condition in such a way that the path most likely to be taken is the fall-through branch path. The dynamic information is used to determine the likelihood of a particular condition evaluating to true, thus facilitating this optimization.

In super-scalar architectures the branch unit tries to resolve upcoming branches while other computations take place in the other units. Sometimes it is not possible to schedule a condition code evaluation early enough to allow the branch unit to completely resolve a branch before the instruction is actually encountered. In some of these cases, frequency information on which branch tends to be taken would be very useful. Knowing which branch is more likely, the compiler could replicate the first few instructions (if semantically sound) of the target and insert them before the branch instruction. This is effectively another version of branch prediction, but exploits a different feature of the RS6000 architecture.

Frequency information can also help in measuring the benefits of certain optimizations under conditions that can only be determined at run-time. For example, in code motion, when instructions are moved outside loops, there is usually some overhead involved with doing so. In the case where the loop is actually being executed a very small number of times (though it could potentially be executed a large number of times) it may be more efficient not to perform the code motion, since the overhead would cause a significant delay relative to the time spent within the loop. Typically when these optimizations are impemented, the loop is assumed to execute for some minimum number of times. So this optimization would be implemented by checking to see if the execution frequency of the loop was indeed more

than the assumed minimum. If not, then the target code would not be modified.

The current implementation with the default profiler provides frequency counts as the feedback data, therefore all the optimizations outlined above are supportable by the interface and the *xprof* profiler together. Unfortunately, time constraints would not allow the implementation of all of these, although branch prediction was attempted. The details of the implementation of branch prediction are given below.

4.3.2 Using Other Types of Feedback Information

Although certainly a useful type of feedback data, the frequency information is not the only type of merit. As mentioned above, if we had data that summarized values of variables throughout the run, we might be able to use this to obtain correlations with procedures called, or memory addresses referenced. One use of such data would be to improve cache performance by reordering instructions or by hinting at cache lines to prefetch. Another use of this type of data is in conjunction with techniques that statically analyze programs to derive symbolic expressions for the probabilities and relative frequencies of edges in the flowgraph such as the one described by Wang in [18].

Since the semantic model used is the control flowgraph, any type of data that is basic block oriented, would be suitable for directing optimizations. In particular, if we had a profiler that returned cache miss information, say, per basic block, then we could implement the optimization described above in the system as is with no need for modification to the interface.

Unfortunately, a limitation of the implementation is that it assumes the data collected is for the control flowgraph. While the drawbacks to this assumption are bearable, it would have been cleaner to have the model that the data applied to specified in the interface. Then we would be able to handle almost any type of feedback information that someone could think of.

4.4 Branch Prediction

Branch Prediction was the only feedback based optimization that was implemented. The compiler assigned probabilities to each edge based on the frequencies obtained from the flowgraph annotation stage by dividing the frequencies on each edge by the sum of the frequencies of edges leaving the same node. Edges were also assigned costs which corresponded to the lengths of the basic blocks plus the inherent cost of traversing the branch depending on whether it represented a true or false predicate of the conditional branch statement. The probabilities on each edge were then multiplied by the assigned costs to yield a weighted cost for each branch. If the weighted cost for a branch was higher than the other by a fixed threshold value, then the compiler rearranged the condition so that the higher weighted cost was on the true branch.

We used an expected cost criterion in light of Fisher's report of better results when "number of instructions per branch taken" was used as a criterion for predicting branch directions instead of just the raw probabilities [11]. Note that we could just as readily have used

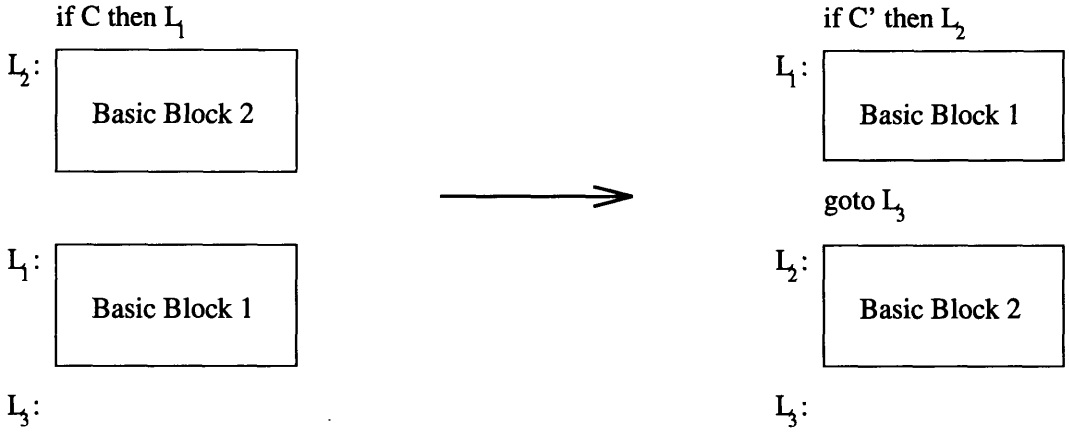

Figure 4.1: Reordering Basic Blocks for Branch Prediction

the raw frequencies as the predicting data, by simply changing the expression for assigning weights to the graph edges in the specification file. This relative ease of adjusting the criterion is a clear advantage to the programmable interface and modular design of the implementation.

4.4.1 Implementing the Transformation

There are two steps involved in this implementation of branch prediction:

1. Negate the condition test for the branch instruction
2. reorder the basic blocks so that the target of the previously true condition immediately follows the branch.

The first step preserves the semantics of the program after the basic blocks are moved and the second step moves the target basic block to the fall through position. After these two steps have been completed, the fall through and taken basic blocks for the branch instruction would have been exchanged. As an example, suppose code at label L_1 is reached when condition C evaluates to true, otherwise the code immediately following the branch instruction (call that point L_2) is reached (see Figure 4.1). Then to reorder the branches, we want L_1 to be located where L_2 is currently, but the code should still have the same semantics as before the transformation. Now, if L_1 is to be placed immediately after the branch instruction then it can only be reached if the condition at that branch instruction, C' evaluates to false. Since L_1 was reached when C was true but when C' is false, then it must be that C' is the negation of C. So we see that the two steps described will produce a semantically correct reordering of the branch target basic blocks.

In the actual implementation, by the time this optimization is to be done, the code has already been laid out. This meant that it would have been difficult to reorder the basic blocks according to the dictated order of the probabilities and costs. Instead, whenever a

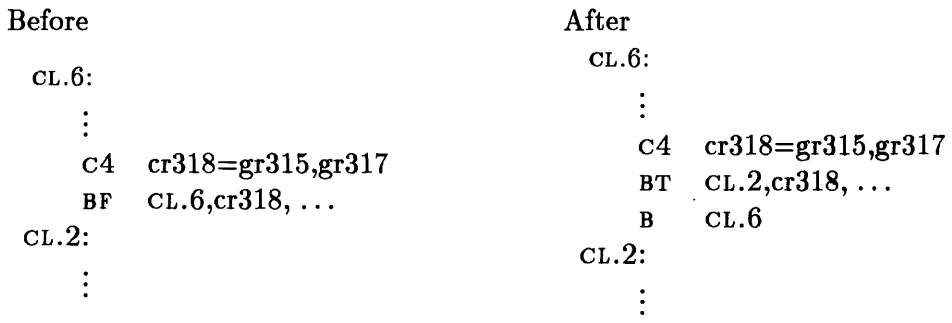

Figure 4.2: Inserting Unconditional Branches to Reorder Instructions

basic block needed to be relocated, an unconditional jump to it was inserted at the point where that code would have been placed — typically immediately after a conditional branch. Figure 4.2 illustrates how this reordering of branches was done. The code fragment is in XIL, the intermediate language of the compiler. BF and BT mean "branch if false" and "branch if true" respectively; B is the mnemonic for the unconditional branch instruction; and CL.n is simply a compiler generated label. The remaining instructions are not important to the discussion, although they indicate the relative positioning of the branch instructions.

The problem with this implementation was that the RS6000 does not carry the conviction of its assumptions in that it will not follow the unconditional branch given that it previously assumed that the fall through branch would be taken. The solution to this problem was to move a few instructions from the beginning of the target basic block (the one that should have been moved had it been easy to do so) to immediately after the conditional branch and unconditional branch. These instructions fill the pipeline so that when the RS6000 encounters the unconditional branch inserted there by the compiler, it can fetch the branch target of the unconditional branch at no extra cost in cycles (see Figure 4.3). The net effect is a reduction in the total number of cycles for the most likely case of the condition code — based on the statistics generated by the input set given in the profiling stage.

4.5 Bibliographic Notes

Fisher originially presented the technique of Trace Scheduling as a method for compacting microcode for a VLIW machine. The larger code fragments permitted more data independent instructions to be identified so that they could be executed in parallel. The wider instructions led to shorter code. This technique has been extended to uniprocessor compilers to allow the optimizer to work with the larger code fragments. It has been demonstrated to have favourable results in refChH,HM in the IMPACT compiler at the University of Illinois where it has been implemented. In [4] Chang, Mahlke and Hwu show how a variety of classic code optimizations can be implemented using frequency feedback information.

Fisher and Freudenberger recently presented some results on branch prediction that indicated that measuring the success of branch prediction can, and ought to be done, by taking the cost of mispredicting branches into account, in addition to the fraction of such

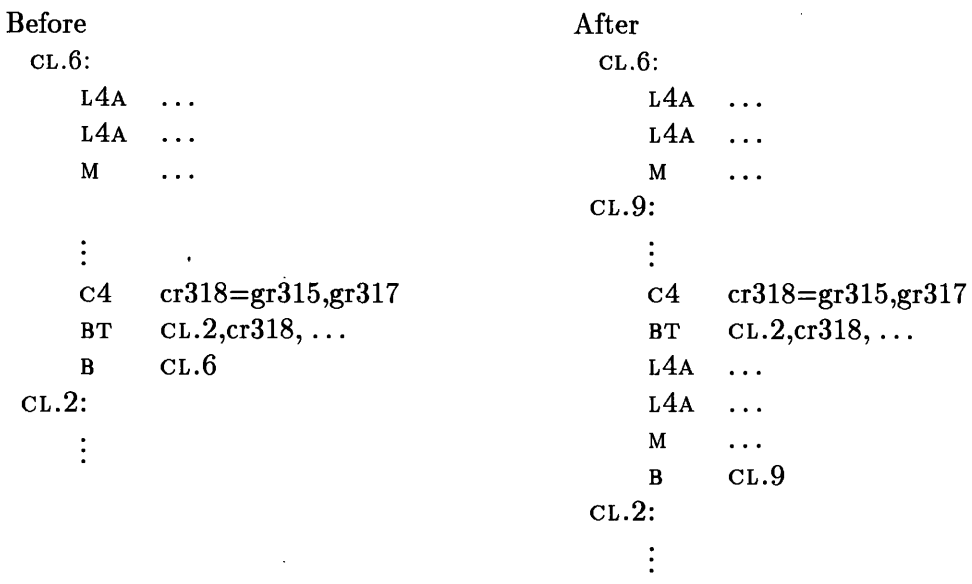

Figure 4.3: Filling the Pipeline between Branch Instructions

branches [11].

The idea of using information about the values of variables in the program was inspired by informal talks with Michael Blair. He proposes to use such information to help direct code specialization for particular data types within languages which are not type checked at compile-time [3].

Chapter 5

A Theoretical Framework for Evaluating Feedback-Driven Optimizations

5.1 Introduction

In this chapter, we use an annotated control flowgraph as the working model for a procedure. We present a general framework for reasoning about how a transformation is directed by feedback data to modify the procedure, and how its relative benefit can be assessed. We also present a technique for annotating the control flowgraph based on execution frequency feedback data and a method for analyzing code transformations based on this technique. This technique is presented in the context of the framework outlined, with specific meanings assigned to the parameters of the framework. We then apply the general technique to branch prediction in particular to show how that optimization would be analyzed using the theory presented.

Ideally, we would like to view a program at two resolutions: the *call graph* and the control flowgraph. In the call graph, each node represents a procedure and an edge from node A to node B exists if procedure A can call procedure B. This graph represents the broad view of the program, and profiler information for this model usually comes from sampling the state of the machine at regular intervals. The relative frequencies with which the edges and nodes are annotated indicate which procedures are called most often and from where. This model would be used first, as a guide indicating which procedures ought to be analyzed further. Deciding that, the control flowgraph would be used to model one of those procedures, where we could then perform the analysis described in this chapter. This ideal situation would require that both granularities of profiled information be available simultaneously, which unfortunately, is not available within this current implementation. However, it is relatively straightforward to manually obtain profiler information on the call graph so the effects of that shortcoming are minimal.

5.2 The Generalized Framework

We characterize feedback based optimization on a deterministic procedure, P, with the following parameters:

1. The set of inputs, I, that P accepts.
2. The set of procedures, A, derivable from semantic preserving transformations on P.
3. The control flowgraph, $G_P = \langle V_P, E_P \rangle$.
4. Cost functions $C_v : V_P \rightarrow \mathbf{R}$ and $C_e : E_P \rightarrow \mathbf{R}$ for the nodes and edges of G_P.
5. The feedback data, $\phi_P = (\phi_P^v, \phi_P^e)$, for both the nodes and edges respectively, where $\phi_P^v : I \rightarrow F_v = \{f \mid f : V_P \rightarrow \mathbf{R}\}$ and $\phi_P^e : I \rightarrow F_e = \{f \mid f : E_P \rightarrow \mathbf{R}\}$.
6. A function to compute the feedback, $\tilde{\phi_P}$, for a set, S, of inputs in terms of $\phi_P(\alpha)$, for each $\alpha \in S$. $\tilde{\phi_P} : \mathcal{P}(I) \rightarrow F_v \times F_e$, where $\mathcal{P}(I)$ is the power set of I.
7. The set of feedback based transformations, T, that preserve the semantics of P. $T = \{\tau \mid \tau : F_v \times F_e \rightarrow \{f \mid f : A \rightarrow A\}\}$.
8. A measuring function, $m : A \times I \rightarrow \mathbf{R}$

The cost functions can represent any innate features of the program that are not dependent on the feedback, but are considered either by some transformations or by the measuring function or both. The feedback data has been divided into two types of functions, one for the edges and the other for the nodes of the control flowgraph. Note that both of these functions result in functions that assign real numbers to the edges or nodes, as appropriate, of the control flowgraph G_P; this models the fact that the feedback is used to annotate the entire flowgraph. Although the resulting functions annotate the graph with only a single real value, we could easily extend the definition to assign a set (vector) of real values, or even values from some other algebraic system other than a field, so long as they can be used in the computation described by m.

The set T, is the set of all the transformations that use feedback information and preserve the semantics of P throughout their transformations. Each of the transformations within T use the annotation of the whole flowgraph to produce new code semantically equivalent to P. The feedback data for a set of inputs, $\tilde{\phi_P}$ is still a pair of functions from the sets F_v and F_e so that it can be used to direct a transformation in T. Therefore, T is not constrained to contain only transformations that base their manipulations on feedback from only one input, although these are certainly contained within T.

The measuring function is a generalization of the various means by which we could assess the benefit of an optimization. In this framework, we will treat the function value of m as one that needs to be minimized. That means that transformations which yield smaller values for m are better than those that yield larger values of m. A good analogy for m is the running time, although, we do not restrict ourselves to considering the running time as the only possible measure for how good an optimization is. One thing to note is that m is defined on a pair of a procedure and an input, but this is slightly deceptive. In

fact m is allowed to depend upon any of the annotations of the graph since they are all dependent upon either the program alone, or the input in combination with the program. The measuring function will typically be in terms of the cost function or the graph and its probabilities or any combination of them. The transformations in T produce programs in A from programs in A, so that the results of the transformations are also valid arguments to m. As an example, suppose that procedure P is profiled with input α, and then transformed with a transformation τ which is based on the feedback results $\phi_P(\alpha)$, then we will say that τ benefits input β if $m(\tau[\phi_P(\alpha)](P), \beta) < m(P, \beta)$.

5.2.1 An Input-Independent Assessment Function

One thing to note is that m is not computed, but rather it is measured, since it involves knowing the bahaviour of a program on a particular input. In other words, although we may write a condition on the relation between the values of m to indicate a beneficial transformation, there isn't any actual way of computing m if it depends upon the input in any non-trivial way, since in many instances such a function reduces to deciding the halting problem. Measuring the value is infeasible, because that amounts to simulating the procedure which can be as bad as actually running it, and therefore unacceptable because of the amount of time it could take.

The basic assumption of the model is that we will be able to compute the probability of traversing an edge given that the procedure has reached the node from which that edge emanates. This probability should be based on the feedback data in some way and will ultimately represent the accuracy of the model. These probabilities will be used to derive a summary function $\tilde{m}$ that represents m "summarized" over all the inputs. In this fashion, we obtain a function which is computable as well as independent of input and hence represents a feasible method of assessing a transformation.

In this presentation we shall use the expected value of m as the summary, however there is no reason that $\tilde{m}$ should be restricted to this type of average value. For example, we could imagine using the most likely path in the graph as the modal analogue to the average instead; or as a slightly more sophisticated value, we could use a weighted mean of the fewest paths that contribute say 90% of the probability of reaching t from s . These modal values would be more useful in the cases of noisy data where highly uncommon inputs that appear in the feedback data influence the mean so that it compensates for them spuriously. A modal analysis ignores these data, so that the assessment is closer to reality.

We have mentioned two possible methods of doing an average case analysis for assessing transformations. There are undoubtedly others that will have merit in some context or other, however the point was to demonstrate how we can go about assessing the relative benefits of transformations on procedures. In this manner, we will have a way of deciding whether a certain optimization ought to be done, without having to actually implement it. This analysis technique should prove to be especially useful for dynamic feedback where the compiler automatically re-optimizes based on new feedback, and so needs to be able to assess transformations without actually implementing them and simulating or observing their running times.

Defining The Summary

We shall use the expression $\Pr[E \mid v]$ to denote the probability that event E occurs given that the current state of computation is at node v in G_P. So $\Pr[v \mid u]$ represents the probability assigned to the edge (u, v) and $\Pr[\alpha \mid u]$ for some $\alpha \in I$ represents the probability that the input α was used given that the computation reached node u in G_P. We also assume the existence of two distinguished nodes s and t in G_P, representing the entry and exit points of the procedure P. All computations are assumed to start from node s and if any computation reaches node t it is considered completed. If there is more than either one of these types of nodes, we can always insert a node of the appropriate type so that it becomes the unique one of that type. For example, if there are two exit points, t' and t'', then we could insert an extra node t and connect t' and t'' to it so that t would be the unique exit node.

The summary function, $\tilde{m}$ is given in terms of m by:

$$\tilde{m} : A \rightarrow \mathbf{R},\ \tilde{m}(P) = \sum_{\alpha \in I} m(P, \alpha)\Pr[\alpha \mid t$$

. Since P is assumed to be deterministic, every input α corresponds uniquely to a (not necessarily simple) path, $p_\alpha = \langle s, v_1, v_2, \ldots, t \rangle$, in G_P, and we can express $m(P, \alpha)$ as a function $\hat{m}$, of p_α, which is computable if p_α is known. So,

$$\Pr[p_\alpha] \mid t] = \prod_{(u,v) \in p_\alpha} \Pr[v \mid u]\Pr[\text{t is reached}]$$

and we get

$$\tilde{m}(P) = \sum_{\alpha \in I} \hat{m}(p_\alpha)\Pr[p_\alpha] \mid t] \tag{5.1}$$

$$\tilde{m}(P) = \sum_{\alpha \in I} \hat{m}(p_\alpha) \prod_{(u,v) \in p_\alpha} \Pr[v \mid u]\Pr[\text{t is reached}] \tag{5.2}$$

5.2.2 Quantitative Comparison of Transformations

Having defined our summary function, we now show how we would use the model to evaluate the merit of a transformation. For any given transformation $\tau \in T$, we assume that τ includes rules for how the probabilities of new edges are assigned based on the original graph annotations, since we do not have feedback data for those new edges to indicate how to assign probabilities to them. Therefore, we can apply τ to P to obtain a new control flowgraph $G_{P'}$ from the resulting procedure P'. Since $\tau \in T$, we know that for the set of inputs, S, used in the profiling, $P' = \tau[\tilde{\phi}_P(S)](P)$ is in A, and therefore the corresponding control flowgraph, $G_{P'}$, could be used to measure $m(P', \alpha)$ for any input α and hence to compute $\tilde{m}(P')$. Hence, we can compare the values $\tilde{m}(P')$ and $\tilde{m}(P)$ to decide whether or not τ is a worthwhile transformation under the input circumstances implied by S. If $\tilde{m}(P') \leq \tilde{m}(P)$ then the expected value of m — the function which defines how transformations are to be compared — for P' over all the inputs, is no worse than that of P, assuming a usage pattern represented by S. In this case, we would conclude that it is beneficial to perform τ on P. Notice that we could extend this further by computing the difference $\tilde{m}(P) - \tilde{m}(P')$ for many different candidate transformations on P. The transformation with the maximal difference would be selected as the one to yield the highest expected benefit.

5.3 Cycle-Latency Based Assessment

Since the majority of compiler optimizations aim to improve the running time of the application being compiled, we shall show how the framework can be parameterized to assess these optimizations. We use an estimation of the number of cycles a basic block will take to execute (the cycle latency) as our representation of the execution time. There are actually two possible types of annotations which we can work with: label the nodes with latencies, or label the edges. Annotating the nodes allows for simpler calculations, but has the problem that it does not distinguish between branch edges which may have different cycle latencies. Annotating the edges does take into account the differences in branch latencies, but has a more complicated analysis. Both methods are presented below, the only difference in the parameters being the use of an edge function instead of a node function. As a side effect of using the parameterizations given below, both models allow us to compute the average running time of a procedure all within the confines of the framework, and the given parameterization.

5.3.1 Parameter Specification

The feedback data is given by the frequencies with which the nodes and edges were traversed. We use the the feedback for a set of inputs, $\tilde{\phi}_P$ and derive the probability for traversing an edge, given that we are currently at the node from which it emanates, by taking the fraction of its frequency of the frequency of the node; that is:

$$\Pr[(u,v) \mid u] = \frac{\tilde{\phi}^e_P(u,v)}{\tilde{\phi}^v_P(u)}$$

. We assume that some reasonable method was used to summarize several profiled runs with several different inputs so that these probabilities will reflect the current usage pattern of the procedure.

Annotating Edges with Costs

The cost functions are specified by estimates on the number of cycles the instructions within a basic block would take to execute. In some machines, for example the RS/6000, the branches also have costs assigned to them, although they are not associated with any code. The cost assigned to an edge is the cost of its destination node summed with its inherent cost. The reason for the difference in inherent costs on branches is that taking one branch may have a larger delay associated with it than taking an alternate one. In the RS/6000, this difference in delay was exploited with branch prediction as described in Chapter 4. The cost functions can be defined in terms of the architecture being used so that architecture dependent properties are captured in the model. The difference of cost in the branch taken in the RS/6000 is one example of such an architecture specific property; but we can also extend the cost function to handle properties such as expected delay for LOAD instructions and some arithmetic operations that may take more than one cycle to complete. By choosing the cycle delays according to the architecture of the machine, we can

improve the accuracy of the assessment, as well as customize the analysis for the underlying architecture.

The value of the measuring function, $m(P, \alpha)$, is the sum of the costs of the edges in p_α, where as above, p_α is the path used in G_P by running P on α. Keeping the same notation, the entry node is named s and the exit node t, given the probabilities and costs on each edge, we define m as follows:

$$m(P, \alpha) = \sum_{e \in p_alpha} C_e(e),$$

We summarize the average running time with the weighted average measurement. This weighted average, $\tilde{m}$ is defined as follows:

$$\tilde{m}(P) = \sum_{s \overset{p}{\rightsquigarrow} t} C(p)\Pr[p] \tag{5.3}$$

where

$$C(p) = \sum_{e \in p} C_e(e)$$

$$\Pr[p] = \prod_{(u,v) \in p} \frac{\phi_P^e(u,v)}{\phi_P^v(u)}$$

We shall show later that $\tilde{m}$ is well defined when all edges have strictly positive probabilities. In the cases, where some of the edge probabilities are zero and that causes the probability on another edge to be one, then we have to be a little more careful. A discussion on this is presented in Section 5.3.2.

Annotating Nodes with Costs

In this model every node is assigned some cost which represents the number of cycles required to execute the code for the basic block represented by that node. The measuring function, m, is defined in a completely analogous manner to the definition for edge annotations, except that we sum the costs of the nodes on a path instead of those of the edges of the path. Likewise for the definition of $\tilde{m}$. Note that the probabilities are still on the edges, as they were before. So only the definitions for m and $C(p)$ need to be redefined:

$$m(P, \alpha) = \sum_{u \in p} C_v(u),$$

where p is the path used in G_P by running P on α.

$$C(p) = \sum_{u \in p} C_v(u).$$

5.3.2 Handling Infinite Loops

The weighted average $\tilde{m}$ is defined in terms of all the possible paths from s to t, therefore we have to take care that it is well defined despite all the cycles possible within G_P. A point

to note is that m is defined as the modeled cost between two distinguished nodes s and t, and we condition our computation of $\tilde{m}$ on t being reached. This should automatically eliminate any inputs that would cause P to loop infinitely since they would all be assigned probability zero, given that node t was reached.

In practice, this is not quite guaranteed, since we don't always terminate in running P while collecting feedback — sometimes the profiler stops collecting information because it has reached its buffering limit, for example. In cases, like these, the feedback information is still useful, but we are not guaranteed to have reached node t in G_P. We can prevent unfortunate infinities in our calculation of $\tilde{m}$ by observing the annotation on the graph before doing any computation to see that there is at least one path from s to t that has been annotated with a non-zero probability. If there are no such paths, then our average costs are assumed to be infinity. To enable comparisons even with infinite values for $\tilde{m}$, we can use different end points within G_P that have non-zero probability paths between them. Doing that would still allow us to assess the effect of transformations on some of the internal nodes of the procedure, though we would have to be careful about which nodes we do evaluate, and how we use that evaluation.

We should distinguish between the case of the infinite loop mentioned above and the case of the closure of a cycle in the graph. Note that in the average case analysis, we compute $\tilde{m}$ as a weighted average of m over all inputs. This means that whenever we have a cycle in the graph on a path from s to t (corresponding to a loop in the procedure P) we must average over the inputs that would cause that cycle to be traversed. Since we do not perform any semantic processing on the graph, we must assume that it is possible to traverse any cycle an arbitrary number of times. That is, we must take the average over a countably infinite number of (finite length) paths. The value of that average is the closure associated with that cycle which is computed as follows: if p is the probability of traversing the cycle and c its cost, the value of the closure is given by:

$$\sum_{n=1}^{\infty} ncp^n$$

If there is some edge on the path that has a probability less than 1 then we are guaranteed that $\tilde{m}$ is well defined since the value of p in expression 5.3.2 is less than 1 and hence that expression converges to:

$$\frac{cp}{(1-p)^2}$$

which is finite. Thus, all cycles that meet the criterion stated have a finite cost, all acyclic paths also have finite cost and there are a finite number of cycles in any graph, hence the average cost computed for $\tilde{m}$ will be finite. We pointed out that we would have to check first, before trying to compute $\tilde{m}$, for the existence of a path from s to t with non-zero probability. Since we condition our probabilities on reaching node t in the calculation of $\tilde{m}$, then any cycle that falls on such a path cannot have a probability of 1 for that cycle. Hence, while computing $\tilde{m}$, every cycle encountered on a valid path from s to t will satisfy the criterion that at least one edge has a probability less than 1. That is to say, we need not worry about inifinite loops so long as there is a path from s to t with non-zero probability.

5.3.3 Computing the Average Cost

In the scheme outlined where the costs were associated with the nodes, we can view the model as a Markov chain with rewards since the calculation of $\hat{m}$ for a path has the Markov property. We can see this esily, since

$$\hat{m}(u_1, u_2, \ldots, u_i) = \hat{m}(u_1, u_2, \ldots, u_{i-1}) + C_v(u_i)$$

so that the intermediate value along the path depends only on the previous value and the current change in cost. The computation of $\tilde{m}$ is simply the calculation of the expected accumulated reward for travelling from s to t. This particular problem happens to be a fairly standard problem in the field and can be computed in a fairly efficient manner (polynomial in number of nodes).

In the case where we use the cost annotated to the edges of the flowgraph, we have to modify our model a bit to be able to apply Markov chain theory since it is defined on rewards associated with nodes. One way to do this is to assign each node the sum of its incoming edges' costs and then perform the analysis on this new graph. Another method is to transform the graph so that the edges become nodes and the nodes become edges. In either method, we can obtain a graph which fits the model used in Markov chain theory with rewards, and so we can use the theory to compute the value of $\tilde{m}$.

5.4 Graph Annotation

Since the summary function relies on the probabilities assigned to the edges of the graph, they determine its accuracy, and hence are extremely important. There are several methods to obtain probabilities for the edges which have varying degrees of sophistication. The method used in the technique for computing expected running time used a simple model for the probability law for each edge. This may suffice for a static feedback setting since the predicting set supposedly contains inputs which are equally likely to occur in actual usage. We could also ascribe a weight to each input in the input set that would be used in the computation of the branch probabilities; this would bias certain inputs of the set according to the usage pattern.

In the dynamic setting, we would probably need a more sophisticated computation that took into account feedback data from the previous runs. Thus, the probabilities would be dependent upon their previous values and the most recent feedback data. This seems like a reasonable starting point for investigating the type of probability law for these edges, since intuitively, we would imagine that programs are used in similar ways with similar inputs over short periiods of time. This seems reasonable, since we typically work on one problem at a time, and so the tools we use in the time we are working on that problem tend to get the same types of inputs. Of course, we also want the system to respond to changes that we make in our usage patterns. Herein lies the problem since we must experiment to decide how to tune our probabilities so that they do not lag too far behind the actual usage, and at the same time, they are not too sensitive to minor changes in usage.

Clearly, there is much work to be done in this area. At the moment, we still do not even know what a suitable control law would be for determining the edge probabilities as

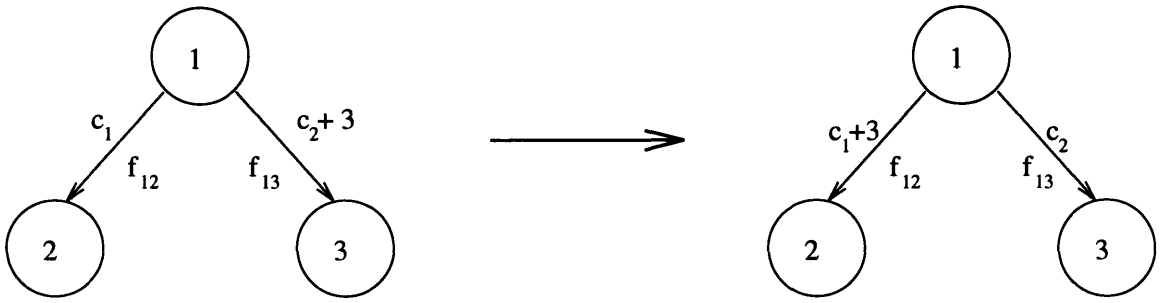

Figure 5.1: The Branch Prediction Transformation

more feedback data is collected. However, there is little we can do hypothetically, and all progress here, is bound to come from actual experiments with new models and with new parameters for old models. This particular aspect of the field should prove to be an exciting an challenging area for research.

5.5 Analysis of Branch Prediction

In this section, we analyze branch prediction on the RS/6000 as a specific example of an optimization to demonstrate the application of the theoretical analysis described earlier. We shall maintain the definition of $\tilde{m}$, the average cycle latency of the procedure P, along with all the previously specified parameters from the description of the technique for measuring average case performance. The only thing left to specify is τ the transformation. In the case of branch prediction, τ is an especially simple transformation since it does not change the structure of the graph, but only rearranges some of the annotations. Figure 5.1 shows the transformation that occurs when a branch satisfies the criterion for being rearranged.

The RS/6000 places a penalty of up to 3 cycles on a taken branch path of a branch instruction. The branch prediction transformation rearranges this penalty assignment so that the branch path that is traversed less frequently is assigned the penalty. The expected value at node 1 is the same for both branches, so the difference in expected value after the transformation is performed, will be due to the difference in expected cost starting from this node. We also assume that both branches will reach the exit node. Therefore the expected decrease in cycle latency will be

$$(f_{12}c_1 + f_{13}(c_2 + 3)) - (f_{12}(c_1 + 3) + f_{13}c_2)$$

which reduces to:

$$3(f_{13} - f_{12})$$

This represents a decrease in cycle latency if $f_{12} < f_{13}$ which is essentially what is done. In the actual implementation, a threshold is needs to be exceeded by the difference in frequencies before the transformation takes place. This threshold, reduces the sensitivity of the transformation to noise in the feedback data.

5.6 Bibliographic Notes

Very little has been pubished on the theoretical aspects of feedback-based compilation. Conte and Hwu [6] presented a theory for measuring the benefit of optimizations in terms of two parameters: the change in the performance due to the optimization and the change in the optimization decisions made as the inputs were varied.

Wang presents a method for annotating the flowgraph using static semantic information combined with algebraic symbolic manipulations, instead of profiling information, to compute probabilities [18].

One method of measuring average execution times based on node costs and probabilities was presented by Sarkar in connection with work done on a parallel compiler at IBM [15]. The method involved first transforming the flowgraph to a tree (details are given in [16]) and then computing the probabilities for the tree so derived.

The ideas for using Markov chain theory to compute $\tilde{m}$ were contribubted in part by Richard Larson and Robert Gallager, independently. The notes for the class they have both lectured recently at MIT written by Gallager present an introduction to Markov chain theory with rewards [12].

Chapter 6

Conclusions

6.1 Evaluation of Goals

The goals of this thesis were two-fold:

1. Design and implement a feedback mechanism, within an existing compiler, in a modular fashion that provided flexible methods for using various type of information in various ways.
2. The second goal was further divided into two parts:
   (a) Investigate the feedback based optimizations possible and any of the special requirements they may have.
   (b) Research the various ways in which programs and their inputs could be characterized so that the feedback-based compilation process could be analyzed in a quantitative manner.

6.1.1 The Design and Implementation

The primary goal of this thesis has been realized. We have shown that it is possible, and indeed feasible, to implement a feedback based compiler in a completely modular fashion. The advantages of such an implementation were made clear by the flexibility in the types of feedback available, and especially by the ease with which we were able to use these different types of data. The disadvantages were primarily a degradation in running time and an increase in the knowledge required to use the system. The latter, we demonstrated can be offset by the use of defaults to facilitate casual users, and the former's effect is reduced, in part, by clever implementations of aspects of the system such as the algorithm presented in chapter 3. We believe that the disadvantages of this approach are outweighed by the advantages, especially in the current state of the art of feedback based compilation, where we are very much uncertain about what types of feedback are relevant and what the best ways to use such informaition are.

One of the problems that arises directly from the modularity of the implementation is that the profiler cannot assume anything about the program it is given to profile, and yet it must try to produce the most general information as possible. In the particular case of trying to produce edge weights for the compiler, we demonstrated that this can be accomplished efficiently by profiling just the node weights. We exploited this fact within the compiler by requiring the profiler to profile only the basic blocks of the program given to it, and then analyzing the graph to determine the optimal placement of instrumenting basic block nodes. In this manner, the cost of the most expensive operation, profiling, was reduced significantly. In our analysis of the algorithm, we obtained interesting results regarding the effects of the graph's topology on the number of edges that needed instrumentation and the applicability of its surprisingly simple greedy strategy which, in some sense, could afford to be oblivious of the topology of the graph.

In addition to our particular implementation, we have identified features which any implementation should have, and we have pointed out ways in which they can be accomplished. In identifying these necessary features, we hope not only to facilitate future implementations and improvements of the current one, but also to establish the salient features of feedback based compilers that would lead to more structured thinking and focussed effort in the area.

6.1.2 The Optimizations

The optimizations that stand to gain from feedback information are numerous, and many were mentioned in the body of this thesis. Although, the time available did not allow us to explore this area satisfactorily, we have argued that the optimizations that can be implemented are highly dependent on the types of feedback information being provided, and on the semantic models used within the compiler. This field, being a relatively unexplored one, has many unanswered fundamental questions. We have mentioned a few: like what kinds of feedback information ought to be considered in optimizations, and what kinds of optimizations would be beneficial and in what ways. Although we do not have the answers to these questions, we have tried to show the connections and interrelations that must exist between them, based on the optimization methods we use today.

6.1.3 Theoretical Analysis

Our secondary goal had two aspects. The first was to explore the optimizations possible, which we have just summarized. The second aspect was to address some of the theoretical issues involved in process of feedback based compilation. This latter part of the goal was a rather ambitious one, mainly because of the scant research in that particular aspect of the field. We have presented a theoretical framework for describing the salient aspects of the process, and have demonstrated how it could be used to assess the benefit of optimizations. Although incomplete, in that we do not have enough applications of it to actual optimizations, we feel that the framework set forth contains the key factors for analyzing any feedback based transformation. By describing the process in formal mathematical terms, we can demonstrate and investigate its various properties. More importantly, though, this formalization helps us to answer some questions which would otherwise have been left up to speculation. For example, in the particular specification outlined in chapter 5, we showed a

method for analyzing the average running time of a procedure. This method in itself can be used as a general device to assess the relative merits of various transformations, assuming the running time (actually number of cycles) is the measurement that we are interested in.

While this framework is quite general, it can be specialized to assess transformations in various ways which are not limited to the typical notions of "faster is better". Although that may ultimately be the case, the particular goal of a transformation may not be to reduce the cycle time by modifying the instructions directly. It may, for instance, be a transformation to reduce the size of the code so that some beneficial side effect takes place (such as fewer cache misses) as a direct result of that transformation. Since there may be many optimizations which will have immediate goals other than reduced cycle time, it is perfectly reasonable not to specify what is being measured for relative assessment of optimizations within the general framework. Rather, we leave it as a parameter to be specified according to the particular set of transformations being compared.

The apparently general use and broad scope of the framework is promising, since it provides us with a means of reasoning about feedback based transformations, and yet does not presuppose anything about the type of data we are using for feedback nor the types of optimizations we intend to perform, nor even how we intend to measure the benefit of the proposed transformations. Of course, having an overgeneralized model prevents us from reaching any non-trivial conclusions, and so is useless. However, we have demonstrated that we can, in fact, specialize some of the parameters of the framework and still be able to reason in very general terms about the transformations we may perform. This demonstrates the viability of the framework since it is general enough to handle probably any feedback transformation, yet sufficiently defined to produce results based on quantitative analysis of the transformation.

6.2 Future Work

Due to the lack of time, there were many aspects of the implementation which could be refined. An example is the user interface. Currently, the specification file contains an expression in terms of the profiler's raw values that represents the data that ought to be annotated to the internal data structures representing the application being compiled. For efficiency reasons, the specification file is read first and that expression compiled and incorporated with the rest of the compiler, so that the expression (which must be evaluated for every line of output of the profiler) will be executed compiled instead of interpreted. This need for the user to recompile the compiler everytime the specification file is changed could be obviated by automating the process. On a more fundamental level, however, much thought needs to be given to the interfacing of the three most important parts of the system: the compiler, the profiler and the user. We have outlined what the necessary features are for a basic interface between the compiler and the profiler, but we have not said much on the user interface, nor have we mentioned how that would be affected by the other interface—as it indeed is.

We have outlined the features we believe to be the keys to a good feedback mechanism, however, many other profilers ought to be used with an implementation of such a compiler to see if these guidelines will truly bear out in the most general of cases. On this matter, our

current implementation, although it can handle various types of information, it can only do so one at a time. Ideally, we would like to be able to glean various types of information and use them all in the same pass of compilation. This might require the use of two profilers or perhaps the same profiler in two different configurations, but it would require an expansion of the amount of data being stored and more importantly, a means of linking the data stored to the relevant data structure. In this implementation, it was assumed that there was only one data structure and that it was the control flowgraph. However, this is not general enough, and ideally we would like to be able to store data for arbitrarily many possible data structures for use with various possible optimizations. An example of this need for annotating more than just a control flowgraph was pointed out earlier, where we mentioned that ideally we would like to have two granularities of representation for the program: a lower resolution one (a call graph, for example) so that procedure interaction could be summarized easily , and a higher resolution one that focusses on the internals of procedures. Both of these semantic models would be useful together when annotated with feedback data and so the feedback mechanism should have the facility to accommodate them both.

6.2.1 The Node Profiling Algorithm

Regarding the algorithm, although a detailed treatment was given in chapter 3, there were still a few points of interest left to investigate. For example, although we know that the topology of the graph determines what the dimension of the right null space of the coefficient matrix, M, will be, we do not have a good way to describe what it means for the graph when we know the dimension of the null space of the matrix. In solving the problem in the algorithm, we work with the matrix and determine its right null space in the end, without ever deducing the properties of the graph itself. The second unexplored area with respect to the algorithm is that we could conceivably improve the performance of the code by performing a second pass of the profiling stage, so that even the profiling could benefit from the feedback. An example is that if a node were assigned a zero frequency count, then every edge connected to it, automatically gets a zero assigned to it. The possible improvements along these lines are explained in detail in chapter 3.

6.2.2 Optimizations

The optimizations possible within a feedback system are certainly among the most interesting aspects of the feeback based system. To start with, all the Trace Scheduling based optimizations which have been implemented before, have been documented to be successes and so should be implemented. Further, some of the optimizations mentioned above have non-feedback analogues which work well and hence show promise for the feedback version. For example, branch prediction has been implemented on non-feedback based compilers using heuristics to determine which way branches ought to be predicted. These have produced relatively good results [2] and so would probably yield net benefits when applied to feedback information.

Some of the other types of optimizations mentioned in chapter 4 involve using different types of information that mere frequency counts. The example of using cache miss data to deter-

mine better layout is an interesting one, and depends on factors that are typically outside of the normal considerations for optimizations. Further investigation also needs to be done on what sort of data should be stored based on the relative benefit of transformations based on that data. That research would also reveal relationships between program structure, program performance and various qualities of the input. It promises to be quite exciting since it poses several fundamental questions which we have no good way of answering offhand.

6.2.3 Theoretical Issues

We have attempted to address some of the theoretical issues involved in the feedback compilation process with the framework presented. Although the example of its application to measuring average runnning time lends credence to the usefulness of the model, we need to show that it can be used for other parameter specifications with equal success. In the description of measuring the average running time, we showed that the average described always existed as long as none of the looping edge probabilities were 1. We also mentioned how that value could be computed, but it is a rather expensive operation and we would like to find more efficient implementations, if indeed they exist.

The specification of the framework presented, itself, is general enough to be applied to many of the optimizations mentioned, as well as some that are not particularly feedback-based, but could be adapted to be so. Note that measuring the average cost need not be as complicated as was specified. However, with the given specification, we model the individual contributions of all the edges in the paths through the graph so that the average so obtained will be quite accurate. Further work in this area could produce interesting alternatives that are easier to compute, and just as representative of the actual costs incurred by executing the procedure.

One aspect of the framework which we did not address in detail, but which is extremely important, is the way in which the graph is annotated. In the case of the given specification, the probabilities that are assigned to the edges will determine how accurate the overall model is. Here, there is a distinction in the type of feedback compilation since the manner of annotation is very different when the application will be continuously profiled and recompiled (the dynamic setting) from when it will not. The dynamic setting is in contrast to when the compiler profiles the code once on a suite of inputs, and based on that suite alone produces a set of probabilities for the edges. Since the latter is the current method of implementation, and it probably will be until machines become powerful enough to be able to reasonably support dynamic feedback, we need research in this area for the short term. However, we also need research in dynamic feedback since the approaches to annotation are different in the two settings. In the static setting, we annotate based on the input suite, keeping in mind that those annotations won't change until the next time compilation is done (manually). In the dynamic setting, we anticipate changing the probabilities assigned and so we will tend to match the current data as near as possible to represent the current usage pattern, whereas in the static setting, we will probably use a more balanced average of all the values. In either case, it is quite clear that much more work needs to be done to determine the appropriate methods for annotation of semantic data structures, such as the control flowgraph.

Appendix A

The Specification File

The Specification file consists of several fields which indicate to the compiler how to interface with the profiler. The fields are:

- The profiler preamble
- The profiler execute command
- Output separators
- The output template
- Switches to decide whether to accumulate frequencies and costs.
- The expressions for computing the weights to be annotated to the graph.

A.1 Profiler-Related Commands

The *profiler preamble* specifies what needs to be done before the profiler is called. In this particular implementation, the *xprof* profiler required that the executable be instrumented first as a separate procceess from the actual profiling. Others require that the executable is generated with certain flags set at compile time, these would be specified here in the preamble. The preamble is a string that will be executed as if it were passed to the shell as a shell script.

The *profiler execute command* indicates how the profiler ought to be called. It is here that the user would specify the mode the profiler should be in (if many are available) and therefore what kinds of data ought to be fed back.

A.2 Parsing The Profiler Output

The *output separators* indicate what characters in the output file of the profiler are separators between the data. Typically, profiler output files are human readable, which means

that the data is usually separated by spaces or some non-numeric character. To preserve generality, no assumptions were made about the format of the output file of the profiler, hence these separator characters had to be specified.

Another aspect of the profiler output format that had to be specified was the *output template.* One assumption about the profiler output that was made was that the data would be presented on a line by line basis. So one entire line was considered to correspond to one record of data; furthermore, the arrangement of the fields within that record were assumed to be consistent from line to line. The output template specified the arrangement of the fields within each line. The output template was a string of names separated by one or more of the characters from the output separators described previously. Here is an example output template:

"freq bbstart bbend tag:file"

This would indicate that the profiler's output consisted of lines with five fields. The first substring on a line surrounded by spaces would be given the name *freq*, the second similarly demarked substring, on the same line, would be called *bbstart*, and so on until the fourth value read on that line. The right separator for that substring would be a colon and it would be called em tag. The rest of the line (after the colon) would be named *file*. Each of these substrings would have some significance as specified by the expressions.

The three fields: *freq, tag, file* have reserved names since these are required fields of all profilers. The *file* field is specified to make the tag reference in *tag* unique, as would have been neccessary in any case, had the tags been line numbers. All other fields specified are peculiar to the profiler being used, and can have any names not matching the three reserved words mentioned above. The substrings within a given line that have been assigned to the names in the output template are used internally either according to the manner indicated in the expression, or if a reserved name, as the compiler was preprogrammed to do.

A.3 Accumulation Switches

While parsing the output file from the profiler, there has to be a policy on how to deal with repeated occurrences of basic blocks. Basic block references may be repeated throughout the output for various reasons depending on the deatils of the profiler. In this particular implementation, some basic blocks, as constructed by the compiler, actually become multiple blocks in the executable and so appear to the profiler as multiple basic blocks. However, they will all have the same tag, so the information is decipherable, but care has to be taken while accumulating over the file, since some values may need to be accumulated while others may not. There are as many switches as there are types of data to annotate the flowgraph with, so in this implementation there are two independent switches: one for the frequencies accumulated, the other for the accumulated costs.

A.4 Computing Annotation Values

The last part of the specification file contains expressions for assigning values to each of the possible annotations to the cotrol flowgraph. In this implementation, we only have two different types of information that can annotate the flowgraph, so we need only two different expressions. The expressions can be arbitrary arithmetic expressions and must be in terms of the names of the fields specified by the output template. So, if the output template had been specified as above, valid expressions are:

*cost_exp = bbend - bbstart wt_exp = freq*cost*

The expressions are evaluated in the order that they are written in the specification file, so the above examples are well defined. From this we see that if *wt* was the data on the flowgraph used to direct decisions, then we could change the criterion being used by simply adjusting the expressions in the specification file.

Appendix B

Invoking The Compiler

Below is the script that is called by the user, which in turn calls the compiler the appropriate number of times with the appropriate options, depending on whether feedback was turned on or not.

plix is the name of the program that calls the compiler proper.

```
#!/bin/sh
#If feedback optimization is turned on, then run compiler in two passes.
#On the first pass, turn off optimization and leave debug information
#in xcoff file (take out -O, -qpredset, and add -g).
#The second pass is simply the original call to the compiler.

#The prefix specifies where to find the appropriate compiler.

echo "$*" | awk '
BEGIN {
prefix="-B./ -tc "
outstr=""
newcmd="-g"
fdbk = "false"
        noexec = "false"
srcfile = ""
}

{
cmd = $0
}

index($0,"-qpredset") != 0 {
fdbk = "true"
for (i=1; i<=NF ; i++)
    if (substr($i,1,9) == "-qpredset")
; # do nothing
    else
if (substr($i,1,2) == "-o") {
   if (noexec == "true")
      outstr = "-qexfn=" substr($i,3)
```

```
   newcmd = newcmd " " $i
}
    else
if (substr($i,1,2) == "-c") {
   noexec = "true"
   outstr = "" #ignore output name
   newcmd = newcmd " " $i
}
    else
if (split($i, fn, ".") >1) {
   if (index("pl8 pl9 plix pl8pln c C f", fn[2]) > 0)
      srcfile = fn[1] #recognized source language
   newcmd = newcmd " " $i
}
    else
newcmd = newcmd " " $i

}

END {
if (fdbk == "true") {
    print("plix " prefix "-qdebug=fdbk1 " newcmd)
    res = system("plix " prefix "-qdebug=fdbk1 " newcmd)
    if (res == 0) {
       system("mv " srcfile".lst " srcfile".lst1")
       print("plix " prefix "-qdebug=fdbk2 " cmd " " outstr)
       system("plix " prefix "-qdebug=fdbk2 " cmd " " outstr)
    }
}
else {
    print("plix " prefix cmd)
    system("plix " prefix cmd)
}
}'
```

Appendix C

Sample Run

To demonstrate how the system would be used, we present a small example in this appendix. Suppose we had the following PL.8 code to compute factorial in a file called `test.pl8`

```
$ep:procedure(inpstr);
dcl
  inpstr char(*);
  dcl
    num integer,
    ;

fact:procedure(n) returns(integer);
dcl
  n integer;
  dcl
    i integer,
    prod integer,
    ;
  prod = 1;
  do i = 1 to n;
     prod = prod*i;
     end do i;
  return(prod);

end procedure fact;

  num = integer(inpstr);
  display("n: "||char(num)||" n!: "||char(fact(num)));

end procedure $ep;
```

Now, if we wanted to compile it using the feedback-based compiler, we must first identify a predicting set of inputs to be used during the profiling of the program. Let us construct a predicting set of 4 elements, say 7,5,4 and 8. The predicting input set file—let us say it is called `testinp`—would look like this:

7 5 4 8

The command to compile the program (branch prediction turned on) involves a call to `tplix` as follows: `tplix test.pl8 -qpredset=testinp`. If the compilation is done with the appropriate compiler options we can get a listing file for the program containing the transformation the code underwent due to the feedback information. Following is a fragment of the listing file which demonstrates the details of the transformation described in chapter4.

⋮

```
** Procedure List #5 for fact Before Branch Prediction **
     8:       HDR
     2:       BB_BEGIN    2
     3:       PROC      gr3=|0
     3:       ST4A      |0(gr1,0)=gr3
     3:       LI        gr369=1
     3:       ST4A      prod(gr1,56)=gr369
     3:       L4A      *gr312=|0(gr1,0)
     3:       L4A       gr313=n(gr312,0)
     3:       ST4A      T)1(gr1,64)=gr313
     3:       ST4A      i(gr1,60)=gr369
     3:       LR       *gr315=gr369
     3:       LR       *gr317=gr313
     3:       C4        cr318=gr315,gr317
     3:       BT        CL.2,cr318,0x2/gt
     3:       BB_END
     3:       BB_BEGIN    3
     4:  CL.6:
     4:       L4A      *gr314=prod(gr1,56)
     4:       L4A      *gr315=i(gr1,60)
     4:       M         gr370=gr314,gr315,mq"
     4:       LR       *gr316=gr370
     4:       ST4A      prod(gr1,56)=gr316
     4:       AI        gr319=gr315,1
     4:       ST4A      i(gr1,60)=gr319
     4:       LR       *gr315=gr319"
     4:       L4A      *gr317=T)1(gr1,64)
     4:       C4        cr318=gr315,gr317
     4:       BF        CL.6,cr318,0x2/gt
     4:       BB_END
     4:       BB_BEGIN    4
     5:  CL.2:
     5:       L4A      *gr314=prod(gr1,56)
     5:       LR        gr3=gr314
     5:       RET       gr3
     5:       BB_END
     5:       BB_BEGIN    5
     6:  CL.7:
     6:       PEND
     6:       BB_END
** End of Procedure List #5 for fact Before Branch Prediction **

** Procedure List #5 for fact After Branch Prediction **
     8:       HDR
     2:       BB_BEGIN    2
     3:       PROC      gr3=|0
```

```
    3:      ST4A     |0(gr1,0)=gr3
    3:      LI       gr369=1
    3:      ST4A     prod(gr1,56)=gr369
    3:      L4A     *gr312=|0(gr1,0)
    3:      L4A      gr313=n(gr312,0)
    3:      ST4A     T)1(gr1,64)=gr313
    3:      ST4A     i(gr1,60)=gr369
    3:      LR      *gr315=gr369
    3:      LR      *gr317=gr313
    3:      C4       cr318=gr315,gr317
    3:      BT       CL.2,cr318,0x2/gt
    3:      BB_END
    3:      BB_BEGIN    3
    4:  CL.6:
    4:      L4A     *gr314=prod(gr1,56)
    4:      L4A     *gr315=i(gr1,60)
    4:      M        gr370=gr314,gr315,mq"
    4:  CL.9:
    4:      LR      *gr316=gr370
    4:      ST4A     prod(gr1,56)=gr316
    4:      AI       gr319=gr315,1
    4:      ST4A     i(gr1,60)=gr319
    4:      LR      *gr315=gr319"
    4:      L4A     *gr317=T)1(gr1,64)
    4:      C4       cr318=gr315,gr317
    4:      BT       CL.2,cr318,0x2/gt
    4:      L4A     *gr314=prod(gr1,56)
    4:      L4A     *gr315=i(gr1,60)
    4:      M        gr370=gr314,gr315,mq"
    4:      B        CL.9
    4:      BB_END
    4:      BB_BEGIN    4
    5:  CL.2:
    5:      L4A     *gr314=prod(gr1,56)
    5:      LR       gr3=gr314
    5:      RET      gr3
    5:      BB_END
    5:      BB_BEGIN    5
    6:  CL.7:
    6:      PEND
    6:      BB_END
** End of Procedure List #5 for fact After Branch Prediction **
```

⋮

Of course, if we were not interested to know what took place, we would never need to generate and look at such a file. We would simply compile the source and run the executable. Note how simple it is for the user to use the feedback information, no prior knowledge of the system nor its internals is required. In fact, there is not even mention of the specification file in the command line since that has been taken care of by the default specification file. The only extra parameter is the name of the predicting input set, which is required under any circumstance, and needs to be user supplied.

Bibliography

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools.* Computer Science. Addison-Wesley, Reading, Massachusetts, 1986.

[2] Ball and Larus. Optimally profiling and tracing programs. In *Proc. 20th annual ACM SIGACT-SIGPLAN symposium on Principles of Programming Languages (POPL)*, All ACM Conferences. The OX Association for Computing Machinery, August 1992.

[3] Michael Blair. Descartes: A dynamically adaptive compiler and runtime system using continual profile-driven program multi-specialization. Submitted to MIT Dept. of Electrical Engineering and Computer Science, December 1992.

[4] Pohua P. Chang, Scott Mahlke, and Wen mei W. Hwu. Using profile information to assist classic code optimizations. In *Proc. Software — Practice and Experience*, 1992.

[5] John Cocke and Peter Markstein. Measurement of program improvement algorithms. In *Proc. IFIP Congress*, pages 221–228, 1980.

[6] Thomas M. Conte and Wen mei W. Hwu. The validity of compiler optimizations based on profile information. In *Proc. 17th annual ACM SIGACT-SIGPLAN symposium on Principles of Programming Languages (POPL)*, All ACM Conferences. The OX Association for Computing Machinery, 1992.

[7] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*, section 17, pages 345–350, 355. The MIT Electrical Engineering and Computer Science Series. MIT Press and McGraw-Hill Book Company, March 1990.

[8] Andre DeHon, Ian Eslick, John Mallery, and Thomas Knight. Prospects for a smart compiler. Transit Note 87, Massachusetts Institute of Technology, MIT Artificial Intelligence Laboratory, October 1993.

[9] John R. Ellis. *Bulldog: A Compiler for VLIW Architectures.* PhD thesis, Yale University, Department of Computer Science, February 1985. ACM Doctoral Dissertation Award 1985, The MIT Press, 1986.

[10] Joseph A. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, c-30(7):478–490, July 1981.

[11] Joseph A. Fisher and Stefan M. Freudenberger. Predicting conditional branch directions from previous runs of a program. In *Proceedings of the 5th International Conference on Architectural Support for Programming Languages and Operating Systems*

(ASPLOS V), number 5 in All ACM Conferences, pages 85–95, Boston, MA, October 1992. The OX Association for Computing Machinery.

[12] Robert G. Gallager. Discrete stochastic processes. Class notes for MIT subject 6.262, January 1994.

[13] Bell Laboratories. Common object file format (coff).

[14] Ravi Nair. Profiling ibm rs/6000 applications. *International Journal in Computer Simulation*, 1994.

[15] Vivek Sarkar. Determining average program execution times and their variance. In *Proc. of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, June 1989.

[16] Vivek Sarkar. Ptran — the ibm parallel translation system. In B. K. Szymanski, editor, *Parallel Functional Programming Languages and Compilers.* ACM Press, 1991.

[17] R. G. Scarborough and H. G. Kolsky. Improved optimization of fortran object programs. *IBM Journal of Research and Development*, 24(6):660–676, November 1980.

[18] Ko-Yang Wang. A framework for static, precise performance prediction for superscalar-based parallel computers. In *4th workshop on compilers for parallel computers*, Delft, Netherlands, December 1993.

[19] Hassler Whitney. On the abstract properties of linear dependence. *American Journal of Mathematics*, 57:509–533, 1935.

[20] W.W.Hwu and S. Mahlke. Trace-based global code optimization. In *Proc. 17th annual ACM SIGACT-SIGPLAN symposium on Principles of Programming Languages (POPL)*, All ACM Conferences. The OX Association for Computing Machinery, 1989.