Die approbierte Originalversion dieser Diplom-/Masterarbeit ist an der Hauptbibliothek der Technischen Universität Wien aufgestellt (http://www.ub.tuwien.ac.at).

The approved original version of this diploma or master thesis is available at the main library of the Vienna University of Technology (http://www.ub.tuwien.ac.at/englweb/).

FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Efficient Profiling in the LLVM Compiler Infrastructure

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Computational Intelligence

eingereicht von

Andreas Neustifter, BSc
Matrikelnummer 0325716

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung
Betreuer: ao. Univ.-Prof. Dipl.-Ing. Dr. Andreas Krall

Wien, April 14, 2010 _____ _____

(Unterschrift Verfasser) (Unterschrift Betreuer)

Andreas Neustifter, BSc
Unterer Mühlweg 1/6, 2100 Korneuburg

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit—einschließlich Tabellen, Karten und Abbildungen— die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, April 14, 2010 _____

(Unterschrift Verfasser)

Contents

Contents **I**

List of Figures **III**

List of Tables **V**

List of Algorithms **VII**

Abstract **1**

Kurzbeschreibung **3**

1 Overview **5**
1.1 What is profiling? 5
1.2 Profiling in the Literature 6
1.3 Goals 8

2 Profiling **9**
2.1 Basics 9
2.1.1 Dynamic versus Static Profiles 10
2.1.2 Types of Profiling Information 11
2.1.3 Granularity of Profiling Information 11
2.2 Methods for Dynamic Profiling 12
2.2.1 Instrumentation 12
2.2.2 Sampling 14
2.2.3 Hardware Counters 15
2.3 Static Profiling Algorithms 16
2.3.1 A Naïve Execution Count Estimator 16
2.3.2 A Sophisticated Execution Count Estimator 17
2.3.3 Estimators for Call Graphs 19
2.4 Dynamic Profiling: Optimal Counter Placement 20
2.4.1 Overview 20
2.4.2 Example 22
2.4.3 Virtual Edges 22
2.4.4 Number of Instrumented Edges 22
2.4.5 Breaking up the Cycles 25

2.4.6 Proof: Profiling Edges not in Spanning Tree is Sufficient 25

3 LLVM **29**
3.1 Overview and Structure . . . 29
3.1.1 Intermediate Language . . . 30
3.1.2 Optimisations and the Pass System . . . 31
3.1.3 Frontends . . . 33
3.1.4 Backends . . . 34
3.2 Why LLVM? . . . 35

4 Implementation **37**
4.1 Used Implementation . . . 37
4.1.1 History . . . 37
4.1.2 Current implementation . . . 38
4.2 Virtual Edges *are* necessary . . . 39
4.3 General CFGs are hard to estimate . . . 40
4.3.1 Weighting Exit Edges . . . 40
4.3.2 Loop Exit Edges . . . 41
4.3.3 Missing Loop Exit Edges . . . 41
4.3.4 Precision Problems . . . 44
4.3.5 Not all CFGs can be properly estimated . . . 44
4.4 How to store a tree . . . 45
4.5 Verifying Profiles . . . 47
4.5.1 Verifying a program containing jumps. . . . 47
4.5.2 A program exits sometimes. . . . 48

5 Results **51**
5.1 Overview . . . 51
5.2 Used Methods . . . 51
5.2.1 Used LLVM Version . . . 52
5.2.2 Used Hardware . . . 53
5.3 Correctness . . . 53
5.3.1 Profile Estimator . . . 53
5.3.2 Instrumentation Algorithm and Profiling Framework . . . 54
5.4 Results Compile Time . . . 55
5.4.1 Profiling Small Functions . . . 57
5.4.2 Differences in Optima . . . 58
5.4.3 Build Times . . . 58
5.5 Results Run Time . . . 60
5.5.1 Runtime Results amd64 Hardware . . . 60
5.5.2 Runtime Results x86_64 Hardware . . . 62
5.5.3 Runtime Results ppc32 Hardware . . . 62
5.5.4 Effectiveness of Profile Estimator . . . 65
5.5.5 Using Profiling Data . . . 66

6 Conclusions **69**

6.1 Future Work 70

Acknowledgements **71**

Literature **73**

List of Figures

2.1 Optimal Profiling Example: Estimation 23
2.2 Optimal Profiling Example: Instrumentation 24

3.1 ϕ nodes select values depending on control flow. 31
3.2 Part of the Alpha backend instruction description 36

4.1 Virtual Edges *are* necessary 40
4.2 Weighting Exit Edges 42
4.3 Loop Exit Edges 43
4.4 Non-estimatible CFG with wrong estimate 45
4.5 ProfileVerifier: ReadOrAssert 48
4.6 ProfileVerifier: recurseBasicBlock 49

5.1 Core part of the SPEC CPU2000 configuration 53
5.2 Percentage of instrumented edges. 55
5.3 Number of Functions with a given Number of Blocks 57
5.4 Instrumented edges per function size. 58

List of Tables

5.1 Percentage of instrumented edges. 56
5.2 Percentage of instrumented edges (sorted by Optimal). 59
5.3 Build times from the SPEC CPU2000 benchmark. 61
5.4 Best runtimes for each SPEC CPU2000 program (amd64). . . . 63
5.5 Best runtimes for each SPEC CPU2000 program (x86_64). . . . 64
5.6 Best runtimes for selected SPEC CPU2000 program (ppc32). . . 65
5.7 Runtimes with and without Profile Estimator 66
5.8 Runtimes with Estimator and Profiling Data 67

List of Algorithms

1 $NaiveEstimator(P) \rightarrow W$ 18
2 $InsertOptimalEdgeProfiling(P) \rightarrow P_i$ 21
3 $ReadOptimalEdgeProfile(P, Profile) \rightarrow W$ 21

Abstract

In computer science profiling is the process of determining the execution frequencies of parts of a program. This can be done by instrumenting the program code with counters that are incremented when a part of the program is executed or by sampling the program counter at certain time intervals. From this data it is possible to calculate exact (in the case of counters) or relative (in the case of sampling) execution frequencies of all parts of the program.

Currently the LLVM Compiler Infrastructure supports the profiling of programs by placing counters in the code and reading the resulting profiling data during consecutive compilations. But these counters are placed with a naïve and inefficient algorithm that uses more counters than necessary. Also the recorded profiling information is not used in the compiler during optimisation or in the code generating backend when recompiling the program.

This work tries to improve the existing profiling support in LLVM in several ways. First, the number of counters placed in the code is decreased as presented by Ball and Larus [19]. Counters are placed only at the leaves of each functions control flow graph (CFG), which gives an incomplete profile after the program execution. This incomplete profile can be completed by propagating the values of the leaves back into the tree.

Secondly, the profiling information is made available to the code generating backend. The CFG modifications and instruction selection passes are modified where necessary to preserve the profiling information so that backend passes and code generation can benefit from it. For example the register allocator is one such backend pass that could benefit since the spilling decisions are based on the execution frequency information.

Thirdly, a compile time estimator to predict execution frequencies when no profiling information is available is implemented and evaluated as proposed by Wu et.al. in [71]. This estimator is based on statistical data which is combined in order to give more accurate branch predictions as compared to methods where only a single heuristic is used for prediction.

The efficiency of the implemented counter placing algorithm is evaluated by measuring profiling overhead for the naïve and for the improved counter placement. The improvements from having the profiling information in the code generating backend is measured by the program performance for code which was compiled without and with profiling information as well as for code that was compiled using the compile time estimator.

Kurzbeschreibung

Unter Profilen versteht man in der Informatik die Analyse des Laufzeitverhaltens von Software, meist enthalten diese Analysedaten Ausführungshäufigkeiten von Teilen eines Programms. Diese Häufigkeiten können entweder durch das Einfügen von Zählern im Programm bestimmt werden oder dadurch, dass der Programmzähler periodisch aufgezeichnet wird. Über diese gemessenen Häufigkeiten für einige Programmteile kann die Ausführungshäufigkeit aller Programmteile berechnet werden.

Derzeit unterstützt die LLVM Compiler Infrastruktur die Analyse von Programmen, indem Zähler in den Programmcode eingefügt werden, die Ergebnisse der Zähler können dann bei späteren Übersetzungen verwendet werden. Diese Zähler werden aber ineffizient und naïv eingefügt, dadurch werden mehr Zähler verwendet als notwendig sind. Außerdem wird zur Verfügung stehende Profilinformation während der erneuten Übersetzung des Programms nicht verwendet.

In dieser Arbeit wird die bestehende LLVM Profiling Unterstützung folgenderweise verbessert: Erstens wird die Anzahl der Zähler, die in dem Programmcode eingefügt werden, auf das Mindestmaß reduziert (Ball und Larus [19]). Dies wird dadurch erzielt, dass, ausgehend von den wenigen eingefügten Zählern, die Ausführungshäufigkeiten von allen Programmteilen bestimmt werden.

Zweitens wird die Analyseinformation so aufbereitet dass der Compiler bei einer Neuübersetzung des Programms diese Informationen verwenden kann. Alle CFG-modifizierenden Teile des Compilers und der Codegenerierungsteil werden angepasst um diese Informationen zu erhalten und zu verwenden. Zum Beispiel kann der Registerallokator die Profilinformation verwenden um die Entscheidung, welche Register in den Speicher ausgelagert werden sollen, zu unterstützen.

Drittens soll ein Schätzalgorithmus implementiert und getestet werden, der während der Übersetzung eines Programms Ausführungshäufigkeiten abschätzt, falls keine Profilinformation zur Verfügung steht (Wu et.al. [71]). Dieser Schätzalgorithmus basiert auf den Laufzeitdaten mehrere Programme, wobei diese Daten mit statistischen Methoden kombiniert werden. Es soll überprüft werden, ob diese Kombination im Vergleich zu der Verwendung einzelner Datenpunkte sinnvoll ist, und ob sie das tatsächliche Laufzeitverhalten des Programms besser abbildet.

Die Effizienz des implementierten Algorithmus zum Einfügen von der Mindestanzahl an Zählern wird evaluiert, indem der Overhead der naïven Implementierung mit dem der Neuimplementierung verglichen wird. Die Verbesserung der Codegenerierung durch Einbeziehung der Profilinformation

wird durch Performancevergleiche zwischen Code, der ohne und mit Profilinformation übersetzt wird, getestet.

Chapter 1

Overview

For the fashion of Minas Tirith was such
that it was built on seven levels,
each delved into a hill,
and about each was set a wall,
and in each wall
was a gate.

J.R.R. Tolkien, "The Return of the King"
(In [51] when referring to system overview.)

This chapter gives an overview on this thesis and on the topics covered. A survey of the available literature and the known algorithms is done and the goals of this work are established.

Chapter 2 provides a detailed look on profiling and on some of the algorithms used later on, it also establishes some nomenclature. Chapter 3 introduces the Low Level Virtual Machine (LLVM) in more detail, the LLVM is a compiler infrastructure that was used to implement several of the profiling algorithms. Chapter 4 covers the implementation of the profiling algorithm in LLVM and discusses some of the more challenging problems and their solutions. Chapter 5 finally presents the results of this thesis: analysis on the algorithms efficiency and measurements on the SPEC2000 Benchmark.

1.1 What is profiling?

Profiling is the act of generating a profile for a piece of software, the term profile describes some sort of characteristic information. As an example the overall execution time for each function in a program (for one execution of the program) is such a characteristic information. As early as 1965 there were attempts to generate runtime profiles for programs, at this time the behaviour of a CPU was recorded by another CPU and written to tape [12].

Donald Knuth first used the term profile when he analysed how often certain FORTRAN statements where executed during the run of the program [52]. In Knuth's case the profile described execution counts per statement. Knuth already anticipated that this kind of information is extremely valuable to programmers, since they can easily pin-point performance bottlenecks. For example the "execution counts per statement"-profile can help the programmer to find the statements that are executed most frequently, usually this statements offer the most potential for runtime optimisations.

Today the term profile is more widely used to describe a certain characteristic that is attached to some part of a program. This could be for example "execution counts per statement", "cache misses per function", "execution time per statement" or some other type of information.

1.2 Profiling in the Literature

The first attempts to collect information on the runtime characteristics of programs were made in the late 1960's by C.T.Apple at IBM with the ambitious goal to record the instruction trace of an IBM 7090 [12]. Although it was not possible to record all instructions they managed to sample enough data to gain an overview on the runtime characteristics of the programs in question. Then the parts of the program where it spent most of its runtime were optimised, this improved the programs runtime significantly with minimal effort from the programmer.

With the advent of compiler optimisations [5, 33, 59] the need to do automated analysis of the programs to assist the optimiser [7, 34] became more apparent. But it was also important to determine areas in which the compiler produced suboptimal code so that new and better optimisations could be found.

This prompted Knuth in 1970 [52] to do a big analysis of FORTRAN code to determine which statements and constructs where used most frequently by (FORTRAN-)Programmers at that time. He developed a program called FORDAP that instrumented a FORTRAN program to record execution counts. When he then optimised the code at the found hot-spots in the program (statements that where executed very often) the execution time of the program was cut down to a fourth of the runtime of the original program.

FORDAP instrumented the whole code, each basic block had a counter attached, this added some redundant counters which resulted in an unnecessary high profiling overhead. This problem was tackled and solved also by Knuth two years later in his seminal paper "Optimal measurement points for program frequency counts" [53] in which he proved how a minimal number of counters can be placed which still provide execution counts for all parts of the program.

Up to this time every part of a program was instrumented, but Knuth instrumented only parts of the code, the profiling information for the other parts was calculated off-line after the program had been run and the profiling information was recorded. Since fewer counters were placed in the program, this also reduced the profiling overhead. In 1981 Forman [42] reduced the overhead even more by placing this counters in parts of the program that were less likely to be executed by describing one of the first static profile estimators.

Also at that time Graham [44] introduced *gprof*, a successor to the widely used Unix tool *prof* (that provided sample based execution time profiling).

The novelty of *gprof* was that it provided not only the raw times which where spent in a method, but it took into consideration the call graph and attributed the time of called methods to the callee. This gave timing profiles a new purpose since it was not only possible to see where the most time was spent but also why this happened, e.g. which callers were using the callee most of the time.

In 1988 Aral and Gertner [13] introduced Parasight which used parallel processors by offloading the profiling process to a separate processor thus adding the possibility to profile parallel applications and reducing the profiling overhead while profiling several other processors. They further reduced the overhead by only selectively profiling parts of the program, using an interactive process to "zoom in" on the areas of interest.

In 1990 Larus [55] tackled the problem of tracing programs where the goal is not only to know how often a statement was executed but also what the previous execution history was at that time. So for each execution of a program the succession of executed basic blocks is recorded, providing hot path information for the functions of the program. Larus did this efficiently by analysing the C-code and placing probes only where the code was non deterministic. This resulted in incomplete traces that were completed by executing the deterministic parts of the program again to provide the full trace, this technique was called abstract execution.

In 1992 Fisher and Freudenberger [41] profiled several big projects and showed that for a given branch the branching probabilities where almost the same for a wide range of input data, suggesting that a well sampled set of input data can be used as branch predictors.

Ball and Larus [18] compared several static branch prediction heuristics in 1993 and used an ordering of these heuristics to improve the overall accuracy, but only Wu and Larus [71] in 1994 managed to combine these heuristics with statistical methods to do accurate static branch prediction.

In 1994 Ball and Larus [19] first implemented Knuth's "minimal number of counters"-algorithm and provided an off-line tool that completed the recorded basic block or CFG edge profiling information. They presented a simple profile estimator that aimed at reducing the runtime overhead of the profiling code by placing the counters in less likely executed regions of the program. They also showed the complexity relations between basic block and edge profiling and that edge profiling is as efficient as block profiling while providing more granular data.

Ball and Larus then also tackled the tracing (path profiling) problem and showed in 1996 [20] how to place trace probes into the program while minimising the overhead. They achieved this by enumerating all unique paths from the functions entry block to its exit and placing code along the paths that added up a number. The resulting number at the end of the function was equal to

the unique number of the taken path, these numbers where stored and could later be used to analyse which paths were taken during the execution.

Anderson [10] showed in 1997 how to profile a whole system by adding sampling code to the kernel and an user mode daemon that processed the kernel-generated data. With this system it was possible to profile all running programs in a running system and to determine which programs and libraries used the most system resources.

Also in 1997 a completely different approach to profiling was chosen by Calder et. al. [27]. They profiled the values of variables trying to determine possible optimisations based on this value information.

1.3 Goals

The goals for this thesis were threefold:

- Implement the optimal instrumentation algorithm for measuring execution counts in LLVM. This includes writing a crude flow-based static estimator that guides the counter positioning, as well as the instrumentation itself and a small helper program to display the annotated code.
- Provide the profile information to the code generating backend, so that e.g. the register allocator can use this information.
- Implement an estimator that combines several heuristics to calculate a static profile that can be used as guidance for the backend in case no dynamic (runtime) profile is available.

Although LLVM already had a small instrumentation framework implemented that provided the base for the new implementation, the old framework was quite buggy and unmaintained, so essentially the whole framework was rewritten from scratch.

Providing the profile information for the backend poses some great difficulties since the control flow graph is modified heavily during optimisation and code generation so maintaining consistent profiling information was and is a challenge.

During the work on the first two points the heuristics based estimator was already implemented by Andrei Alvares [8], so this thesis only contains a discussion of the algorithm.

Chapter 2

Profiling

Therefore whosoever heareth these sayings of mine, and doeth them, I will liken him unto a wise man, which built his house upon a rock.

Matthew 7,24
(From the King James Version of the Bible.)

This chapter establishes the basics of profiling and introduces some important algorithms. In Section 2.1 the basics of profiling are discussed, Section 2.2 presents the methods for recording profiles during the runtime of a program. Section 2.3 covers some of the static profiling algorithms and finally Section 2.4 presents the algorithms used for dynamic profiling.

2.1 Basics

Profiling describes acquiring certain information from a program, this information is called a profile. Depending on the type of profile this information can, for example, be used to

- improve the program specifically in the areas shown to be problematic by the profiling information,
- determine the test coverage of the input data and/or
- provide different input to the program so that different paths in the control flow graph are executed and tested.

The profile information can be associated with certain parts of a program, with functions, call graph edges, basic blocks or control flow graph edges.

The following sections describe certain aspects of profiles, Section 2.1.1 explains the difference between dynamic and static profiles. Section 2.1.2 takes a look at different types of profiling information and Section 2.1.3 covers the granularity of profiles.

2.1.1 Dynamic versus Static Profiles

With *dynamic profiling* the information is recorded during the runtime of the program. A profile can contain information from several different executions of a program. With *static profiling* the information is obtained purely by analysing the program, (e.g. during compile time).

Dynamic Profiling

Dynamic profiling is more accurate than static profiling, since it does not rely on estimates but accurately captures the information when the program is executed. On the down side this approach imposes a runtime overhead, the program runs at lower than usual speed because capturing the profile also needs some of the runtime resources. This can be especially problematic for real-time applications.

Another disadvantage is that the recorded profile is dependant on the input used to run the program. When the used input is not a representative sample of the possible real-world inputs it is likely that the profile does not accurately capture the average runtime behaviour of the program. This can be partly overcome by combining the profiles of several executions with different input data into a single profile.

Static Profiling

In contrast to dynamic profiles, which are obtained by running the program and measuring certain characteristics, static profiles are determined by algorithmically analysing the program (without executing it). For some types of profiling information (e.g. cache misses) this is hard or impossible to do.

Since those static profiles usually are only estimates, they have different properties than dynamic profiles:

- Most interesting programs show a non-deterministic behaviour, that is their execution depends on some external factors like input or time. For non-deterministic programs a static profiling algorithm can only provide relative values (for a discussion of relative and absolute profiles see Section 2.2.2). Although it is theoretically possible to calculate absolute profiles for deterministic programs, the required data flow analysis is sometimes hard to do, thus making absolute static profiles impractical.
- Static profiles are not dependant on input, if the profiling algorithm is deterministic, then the static profiling information is deterministic too.

Often static profiles are used during the compilation of programs to help the compiler with certain decisions. E.g. the complier can use an execution count

estimate to help the register allocator make its spilling decisions.

For each algorithm a measure of efficiency and correctness can be established. This measures can then be used to compare algorithms and may help choosing the right algorithm for a given task. This also holds true for static profiling algorithms (often called estimators in the remainder of this work). For estimators the most important measure is, how well the produced estimates are. This can be determined by generating a dynamic profile with the same characteristics as the static one (relative/absolute, type, granularity) and comparing those two profiles.

2.1.2 Types of Profiling Information

There are many types of profiling information that can be derived from a program. Some of the common profiling types (amongst others) are:

Execution Counts For a given part of the program it is recorded how often this part was executed. This is one of the easiest profiles to obtain, although when done inefficiently it poses a considerable runtime overhead.

Execution Times This records how long the CPU spent executing a given program part. It is difficult to capture this information accurately since the measurement itself needs some of the CPU time and thus influences the measurement.

Cache Misses, Number of Branch Mispredictions,

Pipeline Stalls This records how often, in a given part of the program, a cache miss/branch misprediction/pipeline stall occurred, this is usually measured with the help of hardware counters.

2.1.3 Granularity of Profiling Information

Profile information associates a certain piece of information (usually a number) with a certain part of the program. Usually those parts of a program are (in order of increasing granularity):

Functions For each function one counter/timer is stored.

Call Graph Edges When a function calls another function an edge in the call graph is added to represent this and profiling information is attached to this edges. So for example not only the number of invocations of a function is recorded but how often each callee invoked the function.

Basic Blocks A basic block is a sequence of instructions without branches. So, given the first instruction is executed all other instructions in the block are also executed.

Control Flow Graph Edges When a basic block ends it either returns the function or branches to one or more basic blocks, those branches are the edges of the control flow graph.

Statement Each statement has profiling information attached to it.

Instruction Usually each statement consists of several instructions. It is possible to measure some types of profiling information on the instruction level.

Depending on the type of profiling information, it may be possible to infer information for a lower granularity item by looking at information of higher granularity. For example the execution count of a function can be derived from the execution count of the first basic block (the entry block) of this function. The execution count of the entry block in turn can be determined by the sum of all CFG edge counts that leave the entry block.

2.2 Methods for Dynamic Profiling

Static profiling is done without running the program, so it does not matter (as long as the profiler terminates) how long this process takes. Dynamic profiling on the other hand is done during the runtime of the program so it is desirable for the overhead this profiling imposes to be as low as possible. (Since usually it is necessary that the software runs at a certain minimum speed the overhead must not exceed a certain level).

The problem of keeping the overhead low while still acquiring accurate profiling information was tackled with different means. When instrumenting the code with counters (sometimes called probes) the number and placement of those counters was optimised. Also the profiling accuracy was traded against lower runtime overhead by using sampling instead of instrumenting the source code.

2.2.1 Instrumentation

Instrumentation describes the process of adding code, that performs the recording and storing of profiling information, to a program.

A Small Example

When the execution frequencies of each CFG edge have to be measured, the program is modified so that upon traversal of such an edge at runtime a counter is incremented. Additionally, at the start of the program, an auxiliary function is called that initializes all the counters. At the end of the program the counters are written to a file, if the file exists already it is common practice to either

add the new values to the already stored ones or to append the new counts, this automatically aggregates counts from several executions of the program into a single profile information file.

Instrumentation in General

The instrumentation itself can be done at several stages during the program lifetime:

Source Code To instrument the source code, all the code has to be parsed and instrumentation code has to be placed at the necessary points.

Compile Time During compile time, before invoking the code generating backend the intermediate representation is instrumented.

Binary Modification The executable binary is directly modified to add the necessary instrumentation code.

Run Time Some frameworks allow the dynamic addition of profiling code during the runtime of a program.

Source code and compile time instrumentation both have the advantage of being machine independent, but this also means that it is harder for them to use processor specific hardware counters. Binary modification on the other hand is inherently machine dependent so it has the advantage of being able to use hardware counters much easier, but porting it to different hardware is much harder.

Also, for source code and compile time instrumentation the source code has to be available to be able to instrument programs. This can be a problem when closed source binary programs have to be analysed, binary modifications do not have these limitations.

Compile time instrumentation has the advantage that all the information necessary to instrument the code is readily available and only a small amount of additional information has to be gathered before the instrumentation. Source code and binary instrumentation on the other hand usually have to analyse the program from scratch to find out where to place instrumentation code before doing the actual instrumentation.

Advantages of Instrumentation

Instrumenting a program and executing the resulting binary gives exact profiles that are reproducible with each renewed execution of the program (provided the input is the same and the program otherwise has deterministic behaviour).

Since the profiling information is exact, it can be also used for test coverage or control flow analysis, since it is possible to tell whether or not a certain CFG path was executed.

Disadvantages of Instrumentation

Instrumentation usually has a larger runtime overhead than sampling (see Section 2.2.2).

2.2.2 Sampling

With sampling the executable binary is not modified to gather profiling information, instead the program is halted and resumed during its execution (usually via timed interrupts) to record various properties of the current state. This halting/recording/resuming has to be done with a precise periodicity otherwise the measured values could be biased towards certain parts of the program.

Sampling is usually done with a profiling program that executes and halts the profiled program as needed and records the measured results but it is also possible to modify the program to directly contain this profiling code. With a multi-core system it is also possible to monitor the program on-the-fly via a monitoring routine that runs on a separate core.

Since the profiling information is only sampled at certain points in time, the profiling information is not composed of absolute numbers but contains relative values instead. E.g. if the real execution count for Function 1 is 10 and for Function 2 it is 120 then maybe the measured counts are 3 and 40. It is not possible to say how often Function 1 or Function 2 have been executed, but it is possible to say that Function 2 was executed approx. 12 times as often as Function 1.

Advantages of Sampling

Sampling does not require the program to be changed, but for interpreting the measurements it is useful to have debugging information available for the binary, or to be able to translate the program with debugging information enabled. Also, with a sensible profiling frequency or when using a second core or processor, the runtime overhead is lower than the overhead of instrumentation (see Section 2.2.1).

Disadvantages of Sampling

Sampling only provides relative information, it is not possible to tell how often a certain event occurred exactly. Depending on the task at hand this may or may not be sufficient: e.g. when it is necessary to find out where a program spends most of its time the relative execution frequencies are suitable. But for determining the test coverage of a function relative counts are not enough to ensure that every edge in the CFG was executed at least once.

Since sampling can miss certain events it is not possible to say that e.g. a basic block was never traversed during the execution of a program. It is just as well possible that the sampling never occurred at the time the block was executed.

2.2.3 Hardware Counters

Hardware counters where first introduced in the early 1990's and by the mid 1990's all major CPU manufacturers (most notably Cray but also Silicon Graphics, Intel, IBM, DEC, SUN and HP [74]) implemented hardware performance counters in their microprocessors. This wide availability of performance counters triggered a wide variety of new dynamic profiling implementations that did not rely on instrumentation but instead used these new hardware counters to measure the performance of programs.

Usually these processors had support for certain types of events such as "cycle executed", "instruction issued", "store issued", "branch mispredicted", "cache miss",...

In most implementations not all of these events could be counted at once due to hardware restrictions. There where one or two (seldom more) counter registers that could be configured to count only one of those events. For example the SGI MIPS R10000 had two counters that could be configured to capture two out of 16 event types [74].

Since most programs run on several different hardware platforms it was hard for application- and tool-developers to use this performance counters in a hardware independent way. Several projects aimed on unifying this hardware interfaces into a common API:

Performance Counter Library (1998-2003) The PCL was the first attempt to create an unified API for accessing the hardware performance counters on several different hardware platforms.

perfctr (2002-current) Linux kernel drivers that present a large number of different hardware counters from different platforms as common interface.

perfmon (2002-current) Initially developed by HP perfmon is a kernel module for Itanium, x86_64 and ppc64 architectures that exposes a common API for those hardware counters.

PAPI (1999-current) The Performance API further virtualises the hardware performance counters by providing a completely system independent API while relying on e.g. perfctr and perfmon to actually access the hardware counters. It not only spans a multitude of processors but also supports a large number of operating systems such as Linux, AIX, Unicos and Solaris.

PerfSuite (2003-current) A project that focuses on making hardware profiling not only work but also easy and reliable to use, it relies on PAPI, perfmon and perfctr for data acquisition and Graphviz for data representation.

2.3 Static Profiling Algorithms

This section covers the algorithms for creating static profile estimations, Section 2.3 covers a simple estimator and Section 2.3.2 a more sophisticated one. Finally Section 2.3.3 discusses intra-procedural profile estimation.

2.3.1 A Naïve Execution Count Estimator

This heuristic was introduced by Ball and Larus in 1994 [19] and is still widely used in compilers today. It estimates edge execution frequencies for a single function, but the results can also be used to perform program wide estimates when the results are propagated along the call graph as described in Section 2.3.3.

The basic principle is this: the deeper an edge is nested in conditional branches, the less likely this edge is executed. Also, an edge that is inside a loop is likely to be executed more often, Algorithm 1 gives a short overview.

The algorithm starts by determining the back edges and loop headers of a function by performing a loop detection algorithm that generates information on the natural loops in a function.

The natural loop (as defined by Aho [2]) of a back edge (v_2, v_1) is defined as

$$nat_loop((v_2, v_1)) = \{v_1\} \cup \{v | \text{there is a directed path from } v \text{ to } v_2 \text{ that does not include } v_1\} \tag{2.1}$$

The natural loop of a loop head $nat_loop(v)$ is the union of all natural loops of back edges ending in v. The definition of natural loops leads to the property

that, if v_a and v_b are loop heads, then the natural loops of v_a and v_b are either disjoint or one is completely contained in the other. This also makes it possible to define loop exits of a loop header:

$$loop_exits(v) = \{(v_1, v_2) | v_1 \in nat_loop(v) \wedge v_2 \notin nat_loop(v)\} \tag{2.2}$$

In a second traversal of the control flow graph the edge and block weights are calculated with the following rules, assuming that loops are executed *loop_mult* times:

1. The incoming weight w_i of a basic block is the sum of the weight of all incoming edges that are not back edges. For the entry block of the function (which has no incoming edges) the weight is assumed to be 1.
2. If the basic block v is a loop head with incoming weight w_i and the number of loop exit edges $n = |loop_exits(v)|$ then each edge in $loop_exits(v)$ gets weight w_i/n.
3. If the basic block v is a loop head then the weight of the block is $w_v = w_i * loop_mult$, otherwise it is $w_v = w_i$. If w_l is the weight of the loop exit edges directly leaving the block and n is the number of other edges leaving the block, then the weight for each of this other edges is $(w_v - w_l/n)$.

So the algorithm assumes that, if the control flow graph splits up into n paths in a non-loop-header block, each outgoing edge is $1/n$ as likely to be executed as the basic block itself. Additionally the algorithm assumes an average number of loop executions and multiplies the likelihood of the loop header and its outgoing edges by a factor *loop_mult*.

2.3.2 A Sophisticated Execution Count Estimator

In 1994 Wu and Larus [71] presented an estimator that combines several predictions regarding the outcome of a branch to make more accurate estimations. The algorithm relies on real world programs that are profiled first, this programs and their profiling data is then analysed and the collected results are used during the estimation of arbitrary programs.

For the analysis several categories were established and the branches in the analysed programs then would fall into one or more of these categories:

- Branches that either take a back edge to the loop head or to a block outside the loop.
- Branches based on comparisons (e.g. between pointers, on pointer is null,...)
- Branches based on the contents of the next basic block (e.g. is one branch target a loop header, does one branch target contain a call, does the branch target return,...)

Algorithm 1 $NaiveEstimator(P) \rightarrow W$

for all functions f in program P **do**
  for all blocks b in function f **do**
    determine $back_edges(b)$ and $is_loop_head(b)$
    determine $nat_loop(b)$
    determine $loop_exits(b)$ edges
  end for
  for all blocks b in function f **do**
    if b is the function entry block **then**
      $w_i := 1$
    else
      $e_i := \{(a,b)|(a,b) \notin back_edges(b)\}$ // incoming edges
      $w_i := \sum_{e \in e_i} w_e$
    end if
    if $is_loop_head(b)$ **then**
      for all $e \in loop_exits(b)$ **do**
        $w_e := \frac{w_i}{|loop_exits(b)|}$
      end for
      $w_b := w_i * loop_mult$
    else
      $w_b := w_i$
    end if
    $e_l := \{(b,c)|(b,c) \in loop_exits(b)\}$
    $w_l := \sum_{e \in e_l} w_e$ // exit edges already have weight
    $e_o := \{(b,c)|(b,c) \notin loop_exits(b)\}$ // outgoing edges
    for all $e \in e_o$ **do**
      $w_e := \frac{(w_b - w_l)}{|e_o|}$
    end for
  end for
end for

Additionally dynamic profiles of the real world programs were measured. This measured values were used to determine the probability that, given a branch falls into one of the categories, the branch is actually taken. This results in several heuristics, for example if a branch is based on a comparison of a pointer to null, the probability that the branch is taken is 60%. Or if the block that is branched to returns the function the probability for it to be taken is 72%.

When analysing a program the algorithm determines for each branch into which of the categories it falls. When the branch falls into more than one category, the possibilities of the heuristics for this category are combined with statistical methods to estimate the overall probability that this branch is taken.

When using those estimated branch probabilities to estimate execution frequencies for all of the control flow edges and basic blocks care has to be taken when the function contains loops. Wu and Larus thus also presented a method to calculate this execution frequencies from local branch probabilities which works for reducible control flow graphs. (Reducible CFGs are graphs where the loop head dominates all blocks in the loop see [71] for details.)

2.3.3 Estimators for Call Graphs

The previous two estimators (Sections 2.3.1 and 2.3.2) are concerned with the estimation of (relative) execution counts inside one function. Having that information it is also possible to estimate the execution counts for functions and for the edges in the call graph of a program, this method is also presented in [71].

When a function f calls a function g multiple times, then the local call frequency $lfreq(f, g)$ is the sum of the execution frequencies of each block b that calls g. The global call frequency (for f calling g) is the local call frequency times the number of invocations of f.

Assuming that $cfreq(f)$ is the number of invocations of f and $gfreq(f, g)$ is the global call frequency of f calling g then:

- if f is the main function: $cfreq(f) = 1$
- if f is not the main function:

$$cfreq(f) = \sum_{p \in pred(f)} lfreq(p, f)$$

-

$$lfreq(f, g) = \sum_{\{b \in f | b \text{ calls } g\}} freq(b)$$

- $gfreq(f, g) = lfreq(f, g) cfreq(f)$

2.4 Dynamic Profiling: Optimal Counter Placement

In this section an algorithm is presented that instruments a program with the minimal possible amount of edge counters.

2.4.1 Overview

When instrumenting a program to measure execution counts, it is possible to simply attach a counter to every edge in the program. Unfortunately this is inefficient and imposes an unnecessary high runtime overhead onto the program.

To get rid of the redundant counters Knuth in 1973 [53] devised a method for only placing counters on certain edges in the CFG. The generated dynamic profile was incomplete (only for edges with an attached counter the execution counts were known) but an off-line algorithm which was running later on was used to calculate the execution counts for the edges which had no counter attached. Knuth was also able to show that his method only inserted the minimal necessary amount of counters. Starting on page 25 the proofs are given that Knuth's method is indeed optimal by showing that the number of inserted edges is sufficient and necessary. Sufficient means that really only those edges are needed to profile the function and necessary means that all of these counters are necessary, when removing one the profiling is not complete any more.

The algorithm (see Algorithm 2) operates on each function by first calculating a spanning tree of the control flow graph. All edges that are *not* in the spanning tree (edges attached to a leaf node) get a counter attached. After the program has been executed the profile is completed by calculating the execution counts for the edges of the spanning tree itself (Algorithm 3). Since each leaf node has edges attached to it that are associated with a counter and thus have profiling values, the execution count for the leaf node itself and the edge connecting it to the tree can be determined.

The runtime behaviour for the instrumented program can be further improved by placing the counters on edges that are less likely to be executed. This can be done by estimating a profile in some way (see Section 2.3) and then creating a maximum spanning tree using the estimated edge weights instead of an arbitrary spanning tree.

Algorithm 2 $InsertOptimalEdgeProfiling(P) \rightarrow P_i$

```
create array C in P
index := 0
for all functions f in program P do
  // calculate the spanning tree for f
  ST := ∅
  for all edges e in function f do
    if adding e to ST does not create a cycle in ST then
      ST := {e} ∪ ST
    end if
  end for
  // add counters to P
  for all edges e in function f do
    if e ∉ ST then
      add code to P such that {C[index] + +} is executed when e is traversed
      add code to P that initialises C[index] with 0
    else
      add code to P that initialises C[index] with −1
    end if
    index + +
  end for
  add code to P that writes counter array to file at end of execution of P
end for
```

Algorithm 3 $ReadOptimalEdgeProfile(P, Profile) \rightarrow W$

```
read array C from Profile
index := 0
for all functions f in module P do
  // read profiling information
  for all edges e in function f do
    w_e = C[index]
    index + +
    // when the edge had no counter attached, add it to open set
    if w_e == −1 then
      O = {e} ∪ O
      w_e = −1
    end if
  end for
  // recalculate counter for edges in open set
  while |O| > 0 do
    for all e ∈ O do
      if either end of e has no adjacent edges in O then
        calculate w_e from weights of adjacent edges
      end if
    end for
  end while
end for
```

2.4.2 Example

In Figure 2.1 a CFG with an optimal edge profiling instrumentation is shown. First Algorithm 1 determines the given estimation of the edge weights then the maximum spanning tree is calculated resulting in the tree with the dashed edges. (The edge $(0, entry)$ and $(return, 0)$ are virtual edges that are required by the algorithm to optimally instrument the CFG (for details on these edges see Section 2.4.3)). The solid edges, the ones that are not in the MST, are fitted with counters.

In Figure 2.2 a measured profile of the program is given. Of course only the solid edges had counters attached, so only these edges are really measured. The other edges are calculated according to Algorithm 3:

- Edge $(bb3, bb5)$ has weight 75, edge $(bb4, bb5)$ has weight 6, so edge $(bb5, bb6)$ has necessarily weight 81. From these two edges also the edges $(bb2, bb3)$ (weight 75) and $(bb2, bb4)$ (weight 6) can be calculated.
- Edge $(bb6, bb2)$ can be calculated from $(bb2, bb3)$ and $(bb2, bb4)$.
- Edge $(bb7, bb9)$ gives the flow for $(bb6, bb7)$, this edge, together with $(bb6, bb2)$ and $(bb5, bb6)$ can be used to calculate $(bb1, bb6)$.
- ...

2.4.3 Virtual Edges

The algorithm assumes that a function has a single entry and exit point, those two points are conceptually connected via a virtual edge. This edge creates a cycle that is also broken by the algorithm (see Section 2.4.5) so either the flow entering or leaving the function is counted, not both.

Since most functions have more than one exiting blocks the implementation adds a virtual node (named "0" in this thesis) and several virtual edges: one from the virtual block to the entry block of the function $(0, v)$ and one for each exiting block $(v, 0)$. For details on the implementation of this virtual edges see Section 4.2.

2.4.4 Number of Instrumented Edges

If $|v|$ is the number of basic blocks in a function, then it is known from graph theory that a spanning tree of this function has $|v| - 1$ edges. Taking the virtual block 0 into account (see Section 2.4.3) the actual number of blocks is $|v_v| = |v| + 1$ and the actual number of edges in the MST is $|v_v| - 1 = |v| + 1 - 1 = |v|$.

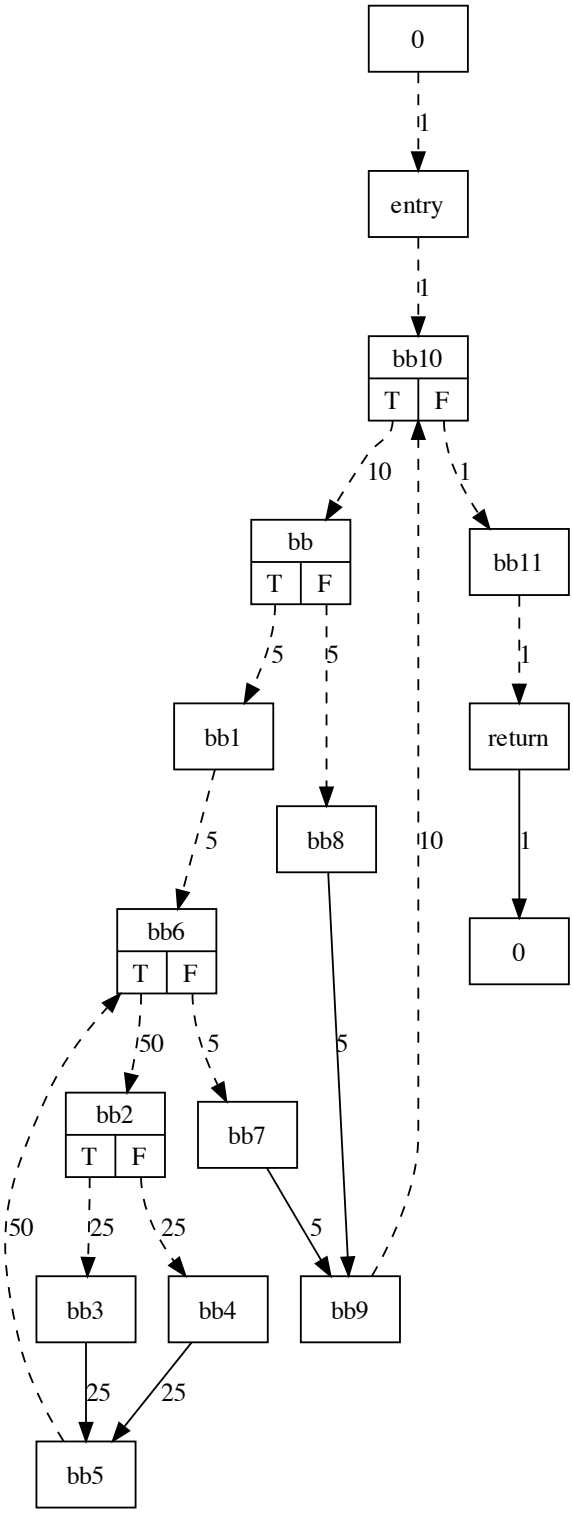


Figure 2.1: Optimal Profiling Example: Estimation

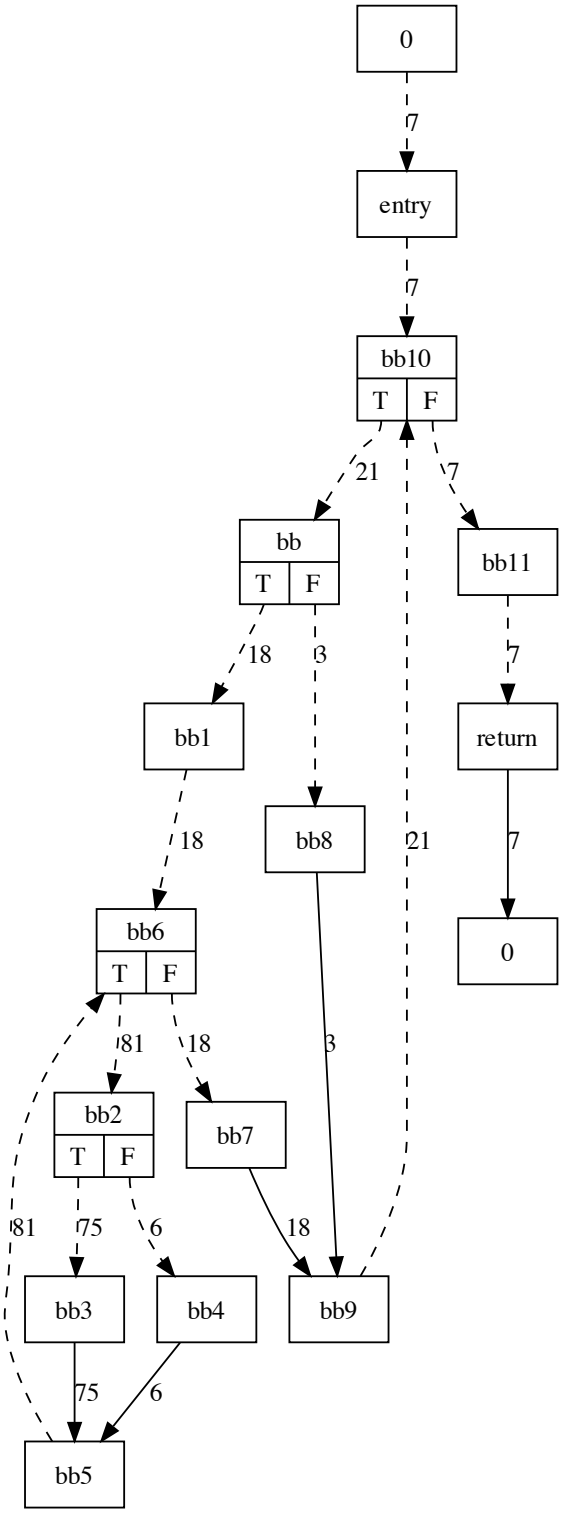


Figure 2.2: Optimal Profiling Example: Instrumentation

$|e|$ is the number of edges in the function and $|e_v|$ is the number of edges including the virtual edges from and to block 0. Since there are $|v|$ edges in the spanning tree and all edges that are not in this tree are instrumented the number of instrumented edges is $|e_v| - |v|$.

2.4.5 Breaking up the Cycles

The optimal profiling algorithm calculates a spanning tree of a CFG to optimally place the edge counters. Each of this instrumented edges connects two leaf nodes of this spanning tree, inside this tree there is an unique path between these two nodes. This unique path, together with the instrumented edge, forms a cycle in the CFG. This leads to two conclusions:

- Each instrumented edge "breaks up" a cycle (if the edge is removed there is at least one cycle less in the CFG).
- The number of instrumented edges is a lower bound for the number of cycles in the CFG. (There are cycles that are formed by two or more adjacent instrumented edges together with a path in the spanning tree.)

Keeping this in mind is useful when later analysing the results of the optimal edge profiling.

2.4.6 Proof: Profiling Edges not in Spanning Tree is Sufficient

This algorithm requires that the function has exactly one entry and one exit block, this is necessary since then the flow leaving the exit block can be assumed to to be the same flow that is entering the entry block creating a virtual edge $(return, entry)$ between these two blocks. (The actual implementation uses a slightly more complicated setup, see Section 2.4.3 for details.)

Since most of the functions in an arbitrary program have multiple blocks returning the function the variation presented here assumes virtual edges from every returning block v to a virtual exit block 0. Also a virtual edge $(0, entry)$ is assumed that connects this virtual exit block with the entry block, flow is allowed to pass over this edge since for each time the function is entered it must be left again. See Chapter 4 how these edges are handled in the implementation.

Because of these edges each block has at least one incoming and one outgoing edge, this makes the algorithm far more easier to understand, verify and implement.

Assumption: It is sufficient to instrument the edges *not* in the spanning tree of a control flow graph (that has no dangling edges) to record profiling

information for the whole CFG.

Proof: Assume the $CFG = (V, E)$ with V the set of all nodes (basic blocks) and $E = \{(v_1, v_2) | v_1, v_2 \in V\}$ the set of control flow edges in this CFG.

A spanning tree ST of CFG is then a maximal, cycle free set of edges from CFG. The edges not in the ST are in another set $NST = \{e | e \in E \wedge e \notin ST\}$. Given weights for the edges in NST it is possible to calculate the weights of all edges in ST while satisfying the flow condition.

To calculate all the edges in ST proceed as follows:

1. Select an edge $e = (v_1, v_2)$ that is currently a leaf edge in ST, that is, the node v_1 is not adjacent to any other edge in ST. The node v_1 is then connected to the tree only via edge e but (due to the virtual edges required) it has at least two adjacent edges, so all adjacent edges other than e must be in NST.

2. Now, since the weights for edges in NST are known, the weight of the node v_1 and of edge e can be calculated, and e can be moved from the set ST to the set NST.

3. Now select another leaf edge in ST and continue with Step 2. It is always possible to select a leaf edge because even if edge e was the last leaf edge in ST, by removing it from ST either $ST = \emptyset$, then the algorithm is finished, or the one edge in ST that was adjacent to e is now a leaf edge.

Proof: Profiling Leaf-Edges is Necessary

This proof is based on the same preconditions as the previous proof (page 25), namely that all nodes have at least one incoming and one outgoing edge.

Assumption: It is necessary to instrument the edges *not* in the spanning tree of a control flow graph (that has no dangling edges) to record profiling information for the whole CFG.

Proof: Assume that there is an edge $e = (v_1, v_2)$ in NST that is not instrumented. This edge has some special properties:

- Since each node v_1, v_2 is adjacent to at least two edges, e is adjacent to at least two edges e_1 and e_2, each on one side.

- Both edges e_1 and e_2 are in the spanning tree ST, otherwise ST would not be a spanning tree since either one of the edges could be added to ST without creating a cycle.

This implies that both nodes v_1 and v_2 have two adjacent edges that have not counters attached, thus the flow in both nodes can not be calculated this preventing at least two edges in the spanning tree from being calculated.

So a counter on each edge that is not in the spanning tree is necessary for calculating the edges in the spanning tree.

Chapter 3

LLVM

LLVM, the Low Level Virtual Machine is a compiler infrastructure that was initially developed by Vikram Adve and Chris Lattner at the University of Illinois in 2000. Chris Lattner was hired by Apple in 2005 to work on LLVM and prepare it to be used in several Apple projects. With the release of Mac OS 10.6 (Snow Leopard) LLVM is one of the supported compilers for Apple's operating system Mac OS X. Together with *clang*, the C- and C++-Frontend for LLVM (which features superb diagnostic messages and a clean API) LLVM is also tightly integrated into XCode, Apple's development IDE.

LLVM is a young compiler when compared to the most popular open source compiler GCC, which was started in 1985 and released in 1987. LLVM is completely written in C++ and highly modular. The frontend parses the source code and converts it into an intermediate representation (IR). All the optimisations are expressed as transformations on this IR, the backend starts with this IR and generates binary code for the supported architectures.

The (comparatively) young code base and the clean architecture make LLVM a good candidate for experimental implementations and for trying out new analysis- and optimisation-techniques. It is easy to hook an additional module into the system with almost no changes to the existing code. The parts of the system are cleanly separated which ensures a shallow learning curve during the first steps in LLVM.

Section 3.1 describes the LLVM and its features in more detail and Section 3.2 explains the rationale behind the decision to base this work on LLVM.

3.1 Overview and Structure

LLVM is a C, C++, Fortran and Ada compiler that has a complex, powerful and easily extendible optimisation system that works on a LLVM specific intermediate representation (IR). The fronted that parses code and generates the

IR is completely separated from the optimisation and the optimisation in turn is completely separated from the backend that generates the machine code. The IR is fully serialisable, so it can be written to and read from a file during every stage in the compilation process. This allows complex scripting of the compiler during all its stages without modifying the compiler itself. It is also easy to write new frontends for LLVM since the source language can be converted to the IR in a crude non-optimised fashion since all the optimisations are done later on.

3.1.1 Intermediate Language

The LLVM intermediate representation (IR) is a hardware dependent program representation in SSA form which can be stored as a set of interlinked data structures in memory, or serialised in a human-readable version or in a space efficient bytecode representation. Most of the LLVM tools accept both serialised IR representations as input files.

Modules, Functions, Basic Blocks

The top entity in the LLVM IR is a module. A module itself consists of global values, external declarations and functions, the functions consist of basic blocks. Each function has a dedicated entry block and each block has a terminator instruction at its end that either branches to other blocks or returns from the function.

SSA Form

The "single static assignment" form requires that a variable is assigned only once, after that assignment the value remains unchanged.

Traditional (changeable) variables are represented in SSA by a succession of unchangeable variables, each time there is an assignment to the traditional variable a new variable is created in the IR.

When the control flow splits up and there are assignments on both control flow paths, the correct value of the variable has to be selected when the two paths join again. Of course the value of this new variable depends on the path taken, this selection is done in SSA with a ϕ node.

A ϕ node is a special command that selects a value depending on the previous control flow. In the LLVM IR the value is selected depending on the predecessor basic block from which the current basic block was entered. In Figure 3.1 a small example can be seen, two variables are used, x and y. In SSA form, because of the single assignment restriction, there are multiple versions

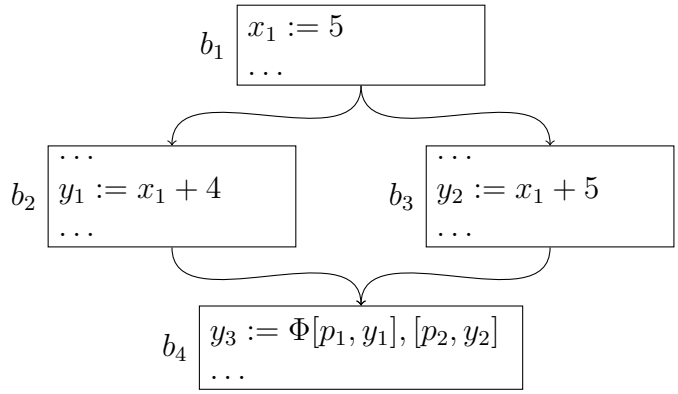


Figure 3.1: ϕ nodes select values depending on control flow.

(index$_1$...) of each variable. When determining the value of y_3 (the third version of y) the ϕ node selects y_1 if the block b_2 was executed before b_4 or y_2 when the control flow came from b_3.

Of course there is no hardware instruction for directly implementing the ϕ nodes but efficient algorithms exist to convert these nodes into a series of copy instructions.

3.1.2 Optimisations and the Pass System

A unique LLVM feature is the pass system. Anything that is done with the IR, loading, analysis, transformation, optimisation, exporting or machine code generation, is a pass. Each pass has a set or properties that help the compiler ordering the execution of this passes:

- Since most passes build on the work of other passes, they can be marked as dependant on other passes.
- Passes can destroy analysis information or revert transformations from previously run passes.
- Passes can also preserve the work from previous passes, but if not noted otherwise a pass is assumed to destroy all information and to not preserve any transformations.

To resolve all this interdependencies between passes (they form a complex dependency graph) and to decide on a scheduling of the passes a so called pass manager is used. When a LLVM tool runs, it tells the pass manager which passes have to be executed, and the pass manager figures out in which order to run these passes for maximum efficiency. Since passes may also destroy previous transformations or analysis results it is common that a pass is run more than once in the resulting schedule.

Another important property of passes is that they operate only on a certain level of the IR. Each pass works either:

- on a whole module,
- on a single function,
- on one loop in a function or
- on a basic block.

One major restriction is posed on the passes: they must not modify any element on the same or on a higher level. For example a function pass is not allowed to modify any function besides the one it is currently running on, a loop pass may only modify the basic blocks inside the current loop.

These restrictions enable the pass manager to schedule passes in parallel. Since a function pass is not allowed to modify anything but the function it was called on, the pass can be run in parallel on several functions.

Analysis Handling

A class can be registered as analysis in LLVM, an analysis is simply some structured additional information that usually is attached to the LLVM intermediate representation. A pass can "implement" this analysis, meaning that if the pass has run this analysis information is available to other passes.

As with passes, analysis information can be preserved or destroyed by other passes, this is also taken into account when the decision is made if already created information is to be used or if the default pass must be scheduled for execution to make the analysis information available again. Some information, like e.g. the dominance frontier analysis is preserved by may passes by informing the analysis class of changes, the class then updates its information accordingly.

In contrast to passes an analysis can be implemented by several different passes, when a pass requests a certain type of analysis the pass manager checks if this analysis is already available because one of the passes implementing the analysis has already been executed. If no pass created this analysis information already, LLVM schedules the default implementation for this analysis.

Most analyses that are used in LLVM provide frequently used but computational expensive information. The analysis system, by caching the results of such analyses, prevents the information from being recalculated over and over again thus reducing the compile time considerably. Examples for this type of information are the analysis that stores information on loops (headers, blocks that belong to the loop, loop exit edges, backedges, ...) or the dominance frontier information. The default implementation for this kind of information is a pass that extracts this information from the IR.

Some analysis information (like the profiling information) is gathered from external sources, this information can not be inferred from the LLVM IR. In this case it is important to preserve and transform the information as the passes are run because the information can not be simply recalculated from the IR. For this type of information the default implementation usually is just a dummy pass that creates some "information not available" information.

3.1.3 Frontends

LLVM currently provides two main frontends: a GCC based one and a newly implemented one called *clang*.

GCC Frontend

GCC has support for a wide variety of languages (C, C++, Objective-C, Fortran, Ada, Java, ...), when LLVM was released in version 1.0 a port of GCC 3.4 was included. This port uses the GCC language frontends and compilation drivers to convert the source to LLVM IR but everything else was done by the LLVM tools. The port was first named C Frontend (because of the lack of C++ support) and later renamed to LLVM GCC Frontend.

With LLVM 1.7 the frontend was ported from GCC 3.4 to GCC 4, but only in LLVM 2.0 the GCC 3.4 port was dropped. In 2007, with LLVM 2.1, a port of GCC 4.2 was introduced and this LLVM GCC Frontend 4.2 is in use ever since.

The (unreleased) GCC 4.5 provides plug-in support that was targeted by the LLVM developers with project DragonEgg, a GCC compiler plug-in that enables the use of GCC as a frontend and compiler driver *without* changing GCC itself, this makes the GCC language frontend easier to maintain.

clang Frontend

clang is a C, Objective-C and C++ frontend for LLVM. The plan to create a dedicated frontend for LLVM that should replace the LLVM GCC Frontend for the C family of languages was presented in 2007. *clang* was designed with a distinctive feature set in mind:

- Support for C, Objective-C and C++. No support for other languages such as Fortran or Java.
- A clean API, so *clang* can also be used as library.
- Support for tracking tokens and macro expansions.
- No frontend optimisations.

- Clear, reliable and useful diagnostic messages.
- Serialisable abstract syntax tree (AST).
- Fast and memory efficient.

The possibility to use *clang* as library, together with the provided token tracking and better diagnostic messages (compared to GCC), makes *clang* especially useful in the context of IDEs. Traditionally, when IDEs used standalone compilers, a piece of code (or the whole program) was handed to the compiler and the resulting messages were read and parsed by the IDE to annotate the code with the compiler error messages.

This communication with the compiler was inefficient, with *clang* the IDE can use the compilation results directly via the API, without having to parse the compiler output. Together with the clearer *clang* diagnostic messages the IDE can annotate the source code faster and more efficiently.

The API, the token tracking and the lack of frontend optimisations also enable more reliable source-to-source translations, it is possible to generate code that is more similar to the input code, with only the minimal necessary amount of changes.

The first release of *clang* together with LLVM 2.6 in 2009 had full support for C and Objective-C. Support for C++ was incomplete but the parser was already able to parse the libstd-C++ library and generated code for simple programs.

clang Static Analyser

Since the *clang* frontend can be used in may different ways to parse code (e.g. for code generation or refactoring) a static analyser was implemented that uses *clang* to analyse source code and to find bugs and problems in the parsed code. The types of analysis that are performed are still in flux but the most used one is memory leak detection. The analyser not only shows potential memory leaks but it is able to explain how this conclusion was drawn by annotating the source code and showing the execution path that leads to problems.

Besides memory leaks, the static analyser is able to find missing initialisations, references to NULL, buffer overruns and dead code.

3.1.4 Backends

When code is generated from the LLVM intermediate representation this is done via a backend, these backends are responsible for generating assembly or object code for different architectures.

Most parts of a backend are not coded in C or C++ but are instead created out of several description files that define the properties of a certain architecture. These description files are processed by an LLVM tool called tablegen that creates C++ code from this descriptions.

This code works by first converting the LLVM IR to a selection DAG, this is a directed acyclic graph that contains all the information of the LLVM IR but also has data flow and control flow analysis attached in a form that is more suitable for code generation.

The instruction selection phase then tries to match the instructions of the requested architecture as efficiently as possible onto the selection DAG and to schedule these instructions to generate the assembly code. After that the register allocator is invoked to resolve the virtual registers to the actual machine registers.

Porting a LLVM to a new Architecture

The first thing to describe when LLVM is ported to a new architecture is the register layout and the associated information like register aliasing and data types for these registers.

In a second step the instruction set is described by describing

- the input and output data types of the instruction
- a small part of the selection DAG with a special graph description language that contains also references to the I/O parameters
- the assembly instruction with the I/O parameters

Figure 3.2 gives a small part of the instruction description of the Alpha architecture.

The important part of an instruction description is that is contains a description of a small part of the aforementioned selection DAG. The operands in this description are nodes of the selection DAG, these operands also appear in the description of the assembly instruction. This links the selection DAG to the instructions.

3.2 Why LLVM?

The decision to use LLVM as a platform to implement and test the ideas and algorithms of this thesis was based on these considerations:

- LLVM is a mature and stable compiler, it has a clean code base and the pass manager infrastructure makes it very easy to add new functionality.

```
1 //Load address
  let
3     OutOperandList = (ops GPRC:$RA),
      InOperandList  = (ops s64imm:$DISP, GPRC:$RB) in
5 {
  def LDA  : MForm<0x08, 0, "lda $RA,$DISP($RB)",
7                  [ ( set GPRC:$RA,
                       ( add GPRC:$RB,
9                        immSExt16:$DISP
                       ) ) ],
11                 s_lda>;
  def LDAr : MForm<0x08, 0, "lda $RA,$DISP($RB)\t\t!gprellow",
13                 [ ( set GPRC:$RA,
                     ( Alpha_gprello tglobaladdr:$DISP,
15                     GPRC:$RB ) ) ],
                   s_lda>;  //Load address
17 }
```

Figure 3.2: Part of the Alpha backend instruction description

- Dietmar Ebner, a colleague of the author at the complang-Institute already had some experience in adding functionality and a tiny part of the problems that this thesis tried to solve was already implemented. This code was not used and all of the functionality had to be rewritten to fit the implementation of the other parts, but it gave important insights on how to do accomplish certain tasks in LLVM.
- The author had a strong commitment to getting his code actually into the LLVM source repository. For that it was necessary to be able to cleanly implement these features without too much tinkering, LLVM provided a great basis for this.
- Last but not least LLVM lacked support in this area and some good profiling was needed anyway, so why not implement it?

Chapter 4

Implementation

> Any sufficiently complicated C or Fortran program contains an ad-hoc, informally-specified, bug-ridden, slow implementation of half of Common Lisp.
>
> *Philip Greenspun's Tenth Rule of Programming*

4.1 Used Implementation

In Section 4.1.1 the old LLVM profiling implementation is explained, in Section 4.1.2 the new implementation that was done during this thesis is described and the differences to the old implementation are highlighted. Of course the new implementation was not perfect from the beginning, the encountered obstacles and their solutions are discussed from Section 4.2 onwards.

4.1.1 History

The LLVM profiling implementation prior to this thesis had the following structure:

There are three passes that instrument a program either on edge, basic block or function level, all three passes can be applied independently to a program to get all three levels of instrumentation. The passes also insert function calls that parse the command line during start up (to read the profile information file name) and that write the counters to the profile file at program termination. If the file is already present on program termination, the new profile information is simply appended.

The profile information file contains several sections of data, each section has a header that gives the type of data and the number of entries. With this simple layout it is possible to combine the data of several executions of the program in one file. The types of data that can be stored in the profile file are

- command line arguments
- function counters
- basic block counters

- edge counters
- path tracing information
- basic block tracing information
- optimal edge counters

(The path tracing information and the basic block tracing information is currently not used since these parts are unmaintained in LLVM.)

The function calls that are added to the program during the instrumentation are implemented in a runtime library that must be linked to the program at compile time. When the program terminated and the profile file was written, the profile file can be used by the optimiser when the program is recompiled.

Since the profile file is a simple stream of counters it is necessary that, during compilation when the profile information is loaded, the program has the exact same structure as when the program was instrumented. This is due to the fact that the program is traversed in a deterministic way, each edge that is traversed gets a value from the stream of counters attached. If the program looks different then also the traversal is different and the association between counters and edges is done wrongly.

The reading of the profile file is done by a pass that also implements the profiling information analysis. The profile information is stored in three flat tables, one for functions executions counts, one for block counts and one for edge counts. After this pass has been executed the analysis can be accessed by other passes that are interested in the analysis.

4.1.2 Current implementation

The current implementation retains the structure of the old implementation and adds features and passes and bugfixes some issues, the main differences are:

In theory edge instrumentation is sufficient to also gain information on the execution counts of basic blocks and functions. Since in LLVM it is possible that a function only consists of one basic block, there are no edges in the CFG of this function and the old edge instrumentation did not instrument the function at all. This created error messages during profile analysis since no profiling information was recorded for this function. The implementation was changed to at least instrument a virtual edge $(0, v)$ where v is the entry block of a function. This ensured that even functions without an edge could be profiled.

The class that stored the profile information analysis was cleaned up. The old implementation used three huge maps (one for functions, basic blocks and

edges) to store execution counts for the program. The maps for blocks and edges were split up into sub-maps. Each of these two maps now contains a sub-map for each function. These sub-maps in turn contain the values for the blocks or edges of this function.

Since the optimal profiling algorithm presented in Section 2.4 uses a maximum spanning tree to improve the optimal edge counter placement, a pass was introduced that produces a crude execution count estimation as detailed in Algorithm 1. This pass presents its results also as profile information analysis to the instrumentation pass.

The optimal edge instrumentation was modelled after the old edge instrumentation pass: First the edges in the program are counted to create a global array in the program that contains a cell for each edge. Then the information from the crude estimator is used to create a maximum spanning tree of the program. Each edge is checked: if it is in the MST the array cell of the global array is initialised with -1, otherwise the program is modified to increment the array cell each time the edge is traversed and the array cell itself is initialised to 0 (see Section 4.4 why these initial values were chosen). The program is then modified to write the counter array to the profiling file (with a new distinct type) at program termination.

As a last step the profile loading pass was adapted to accept the new type of profiling information and to calculate the missing profiling information during the loading of the file: All the counters are loaded, for counters that are -1 the edge is added to a list. Counters that are greater than -1 are associated with their edge and added to the profiling information. After all the counters are read, the execution counts of the edges in the list are calculated according to Algorithm 3.

4.2 Virtual Edges *are* necessary

The old implementation had the problem that functions with only one basic block received no edge instrumentation because there are no edges in the CFGs of such functions. This was solved by implementing a virtual edge $(0, e)$ from a virtual block 0 to the entry block of a function.

This edge was only available in the profile analysis so extra checks had to be performed each time the code iterated over the CFG of a function. To prevent extra checks, initially there were no additional virtual edges implemented for returning blocks (blocks that had no successors). This, as the algorithm suggests, soon proved to be non optimal since all edges leading up to this block got a profiling counter attached during instrumentation. Implementing the virtual exiting edges $(v, 0)$ reduced the number of instrumented edges by approx. 5%.

Figure 4.1 shows a part from the graph in Figure 2.2 as an example of

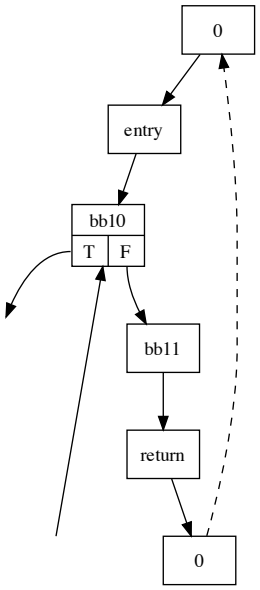


Figure 4.1: Virtual Edges *are* necessary

this situation. Since the MST algorithm actually treats the two blocks 0 as the same block, it only instruments one of the edges on the cycle $(0, entry, bb10, bb11, return, 0)$ instead of two.

4.3 General CFGs are hard to estimate

4.3.1 Weighting Exit Edges

The naïve static estimator as shown in Section 2.3.1 does not work on arbitrary graphs since it assumes that all loops in the CFG are natural loops. A loop usually consists of:

A Loop Header The basic block where the decision is made whether or not to execute the loop (again).

Backedges One or more edges going from the loop back into the header.

An Exit Edge The edge that is traversed in case the loop is not executed any more. In the case of natural loops this edge must be leaving the loop header.

The algorithm works with loops that have multiple backedges, but edges leaving the loop from anywhere else than the header present severe problems.

Consider the loop starting in $bb10$ in Figure 4.2, it has two exit edges $(bb2, bb4)$ and $(bb10, bb11)$. The flow that is entering the loop head must leave the loop, according to the algorithm in its strict implementation all the flow is using the edge $(bb10, bb11)$. But for loops that have no exit edges leaving the head this is non applicable.

So the best version is to assign the weights to all exiting edges (also the ones that leave the loop from the loop body). But then a second problem arises: edges that have flow attached to them may not be changed again since they might have been used to calculate other flow. So each edge gets its flow determined only once, after that the flow must not be changed again.

The loop starting at $bb6$ has two exit edges $(bb2, bb4)$ and $(bb6, bb7)$. Since the edge $(bb2, bb4)$ was already set from the outer loop (during the processing of the loop header at $bb10$) all flow coming into $bb6$ is leaving the loop at $(bb6, bb7)$.

The fixed setting of edges leads yet to another problem that is discussed in the next section.

4.3.2 Loop Exit Edges

An important point with loops is that the weights of the loop exit edges are set when the header is processed because otherwise, since the flow is multiplied in the loop, more flow would leave the loop than entering it. This makes it necessary to ensure that enough flow is reaching this exit edges, there are CFGs where the flow is split up so many times inside the loop that, without special measures, not enough flow is entering the (already set) exit edge.

Figure 4.3 shows a such a loop. When the loop header $bb1$ with incoming flow 1 is processed both exit edges $(bb1, return)$ and $(bb7, return)$ are set to 0.5. The flow entering the loop is multiplied by 5 and split up four times so that not enough flow is entering the innermost block $bb7$ compared to the flow that is already leaving via edge $(bb7, ret)$.

To prevent such wrong estimates each edge has a so called "minimal weight" attached, the weight of the edge must be greater or equal to this minimal weight when the edge is estimated. When the header of a loop is processed and the exit edge e is set to w the minimal weights along a path from the header to e are incremented by w. This ensures that enough flow is reaching the exit edge.

4.3.3 Missing Loop Exit Edges

Every block inside a loop has (directly or via other blocks) the loop header as a successor, so every block in a loop has at least one successor block. Because of this, loop blocks will never get a virtual edge $(v, 0)$ attached.

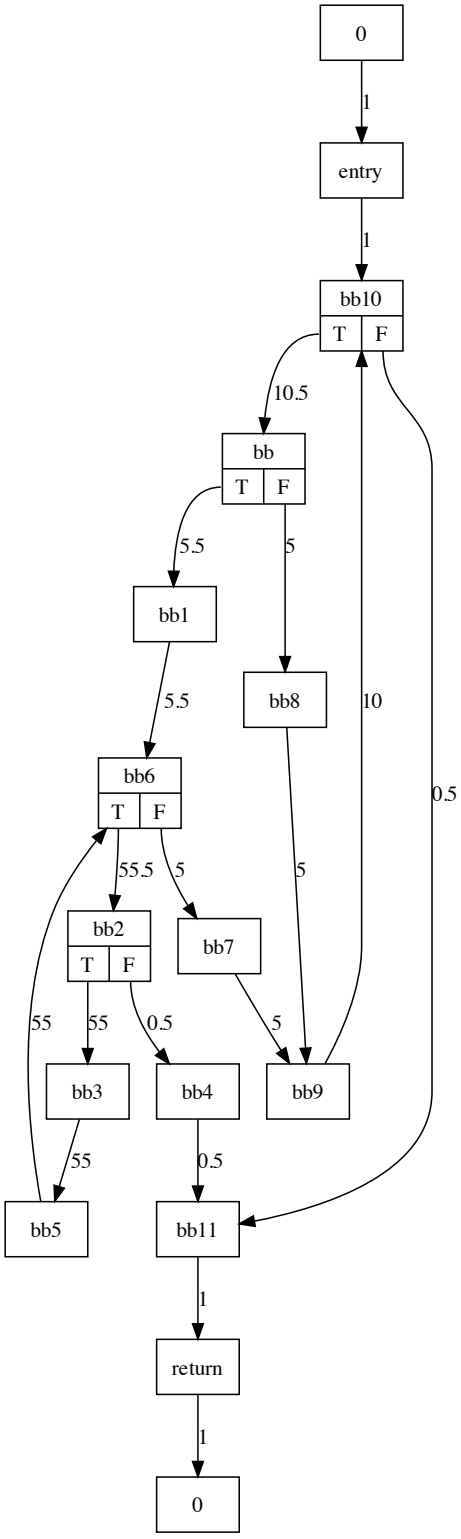


Figure 4.2: Weighting Exit Edges

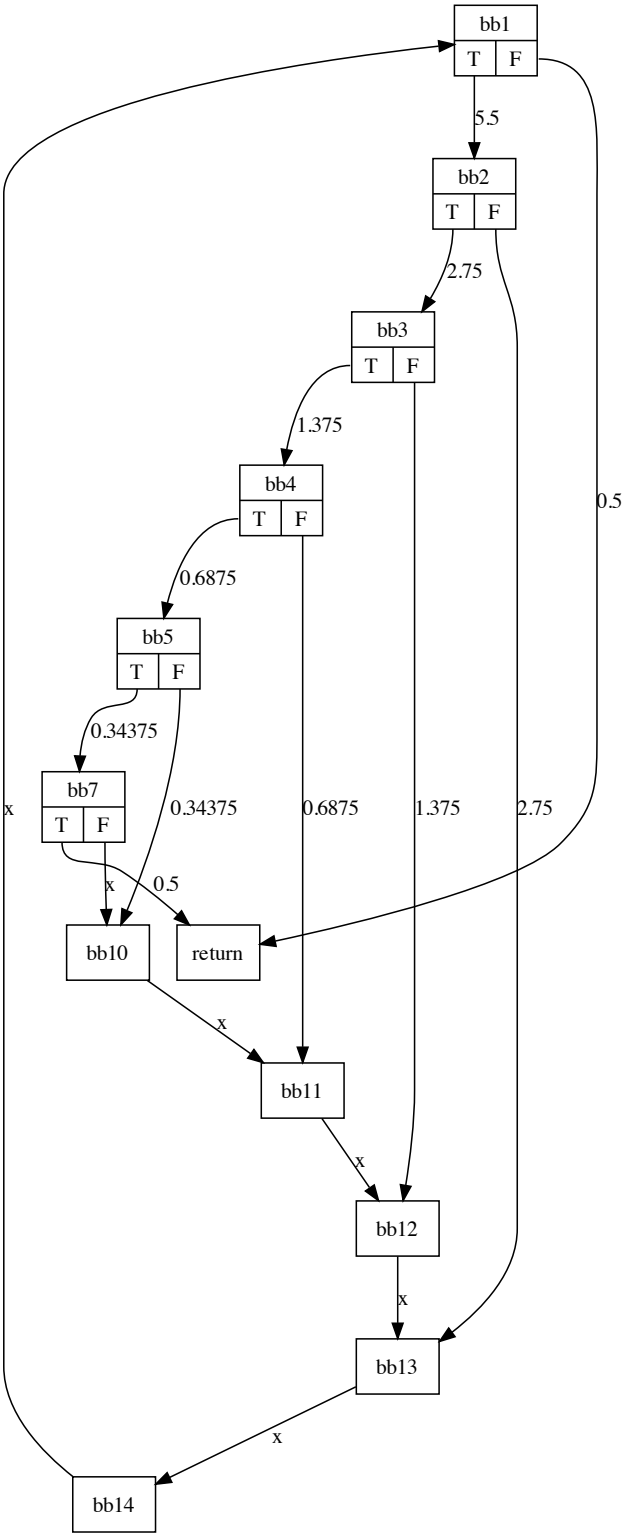


Figure 4.3: Loop Exit Edges

For loops that have not exiting edges this poses a problem because the flow entering the loop can not leave via loop exit edges or virtual edges. In those loops an additional virtual edge $(v, 0)$ is attached to the loop latch v so that the flow can leave the loop properly.

(The latch block of a loop is usually the block that decides whether or not to execute the loop again, in this case the latch has lost all edges branching outside of the loop during the previously executed optimisations.)

4.3.4 Precision Problems

Initially the profile estimator assigned the weight 1 to the function entry block and distributed this weight from there by splitting the weight equally at branches. This leads to very small weights when the block is deeply nested or e.g. a switch statement distributes its incoming weight onto many outgoing edges. When such a weight is added to a really big weight then precision is lost leading to a violated flow condition in some node of the graph.

The solution to this problem was twofold, first of all the weight that the algorithm started with for each function was not assumed 1 but 2^{32} so the value range for a 32-bit value was used optimally. Also the flow of a block was not distributed equally to all edges of the block but the incoming flow v_{in} was distributed to n outgoing edges as follows:

$$v_0 = v_1 = \ldots = v_{n-2} = \lfloor \frac{v_{in}}{n} \rfloor$$

$$v_{n-1} = v_{in} - \sum_{i=0,\ldots,n-2} v_i$$

This ensures that every edge gets assigned an integer value and the last edge gets the remainder, thus preventing rounding and precision errors from creeping in.

4.3.5 Not all CFGs can be properly estimated

As soon as a control flow graph contains non-natural loops it is possible that the graph is not fully estimatible by the algorithm. This occurs for example when not all the loops entry edges enter at the loop head but some of them enter in blocks inside the loop. The loop then can not be recognised as such and the algorithm can not know what to do with the backedges.

Consider the graph in Figure 4.4, there is one loop starting at $h1$ and containing the blocks $l1$, $l2$ and $l3$. The entry block *entry* not only branches to the loop head $h1$ but also right into the loop block $l1$, so the loop can not be recognised as natural loop.

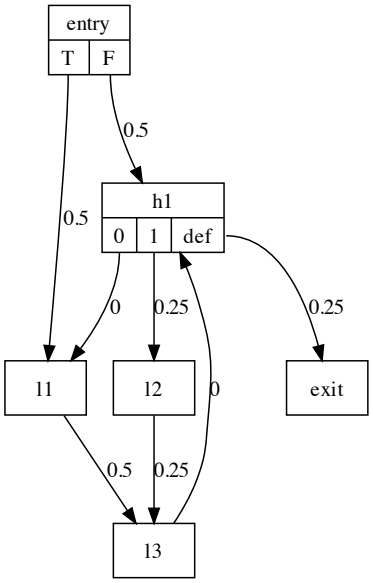


Figure 4.4: Non-estimatible CFG with wrong estimate

The estimator starts at block *entry* and estimates both exiting edges with weight 0.5. When trying to estimate $h1$ it can not proceed since the edge $(l3, h1)$ is not calculated yet, and the loop information does not tell that this is a backedge that can be ignored for now. So the algorithm tries to estimate $l1$ but the edge $(h1, l1)$ also has no value, so the flow passing $l1$ can not be estimated. This deadlock situation must be resolved somehow.

The estimator has a fall back solution that allows to assume that the flow on an uncalculated edge (v_1, v_2) is 0, but only if there is no path from v_2 to v_1. This path-condition is necessary since it is necessary to ensure that the flow on the edge is not self dependant. If this condition is not observed the wrong weights in Figure 4.4 are calculated.

It would be possible to estimate these CFGs with expensive backtracking techniques, but the costs are not worth the effort for such a simple estimator.

4.4 How to store a tree

During naïve instrumentation each edge gets a counter attached, all the counters are stored in a linear fashion in one big array, this array is dumped to a file when the program has terminated. In this file there is no explicit association between the counters and the edges in the program. The counters are implicitly tied to the edges in the order in which the edges are traversed when

traversing the program.

When optimally instrumenting a program the calculated maximum spanning tree is used to determine which edges to attach a counter to. If only these counters are stored to the file a problem arises when the profiling information is read back again: not only the program must be exactly the same, but also the used spanning tree must be the same that was used to instrument the edges, otherwise the association between counters and edges is done wrongly.

A MST is created by sorting the edges by weight and then adding edges that create no cycle in the MST, starting with the heaviest. Since the naïve estimator attaches the same flow to several edges the sorting of this edges is not unique, thus also the MST is not unique. But when the MST is not unique then the loading of the edges can go wrong because two different MSTs are used, one for instrumenting and another one for loading the values from the profile file. Several techniques were tried to ensure that the same MST that was used for the instrumentation was also used during reading the profiling information:

Providing an estimate with unique edge weights. The first attempt in solving this problem was highly unsuccessful. The idea was to alter the estimation algorithm as to not split the flow equally amongst edges but instead to ensure that all the outgoing edges of a block get a different weight assigned. This, first of all was highly difficult to implement correctly and secondly the estimate still assigned the same edge weight to several edges, thus not curing the problem.

Predictable edge sorting. Since the MST is different for the same CFG only if the edges are sorted differently the second attempt was to provide an unique sorting for edges, so that the MST is unique too. This approach looked somewhat reliable at first, but proved to be unusable on general graphs. The problem was that the edges hat to be sorted not only by weight but also by some other distinct features, but due to the nature of the LLVM intermediate representation there is no such thing as an unique basic block. All the features that are unique during runtime (name, position in memory) could change when the same IR is loaded on two different occasions, only the position of the basic block in the CFG is unique. This unique position would have been a solution to the problem but only at the additional expense of calculating and storing these unique numbers before calculating the MST.

Storing the association between counter and edge inside the profiling file. Given an unique numbering of the edges (that is possible by doing a fixed traversal of the CFG) it would be possible to store the unique number of the edge that each counter belongs to in the profiling file. This would, given that the optimal instrumentation instrumented approximately 50% of the edges give a file size that is roughly the same as with naïve instrumentation.

All these attempts had one goal, to reduce not only the number of instrumented edges but also reducing the file size of the profile file by storing only the

necessary counters. It emerged that the cost of reducing the file size came at the price of far more complex algorithms and/or increased compile times. Since file size is not as much an issue as complexity and runtime, a fairly simple solution was chosen finally:

Marking unused counters. When the program is instrumented and the array is prepared for the counter values, *each* edge gets its own cell in the array. For edges that get a counter attached (because they are not in the MST) the counter value in the array is initialised to 0, for edges that have no counter attached, the value is set to -1. Thus during loading of the profile it is trivial to determine which edges had a counter attached and for which edges the execution frequency still has to be calculated.

4.5 Verifying Profiles

Shortly after starting to work on the LLVM profiling support it was apparent that there was a need to verify the profiling data after the estimator and after reading in the profile files. For this a pass was written that took the profile information analysis (provided by the estimator or the profile information loader) and verified that

- for each edge/block/function a value was stored or could be derived from other information in the profiling analysis information.
- for each basic block the sum of the incoming weights was equal to the sum of the outgoing weights (flow condition).

Figures 4.5 and 4.6 show a stripped down version of the profile verifier core.

The profile verifier was an important tool for testing the profile estimator, it found all the problems mentioned in Section 4.3. Verifying the flow condition of the read in profile information was a little challenging though, the following two Sections deal with these issues.

4.5.1 Verifying a program containing jumps.

When a program uses `setjmp` and `longjmp` to control program flow the incoming weight of a block can be smaller than the outgoing weight when the block is the target of a jump. The profile verifier thus detects if the `setjmp` function was used in a block, in this case the incoming weight, and the block weight which is derived from the incoming edges, is not verified.

```
1 double ProfileVerifierPass::ReadOrAssert(Edge E) {
    double EdgeWeight = PI->getEdgeWeight(E);
3   if (EdgeWeight == ProfileInfo::MissingValue) {
      // Assertion Code ...
5     ...
      return 0;
7   } else {
      if (EdgeWeight < 0) {
9         // Assertion Code ...
          .. .
11        return 0;
      }
13    return EdgeWeight;
    }
15 }
```

Figure 4.5: ProfileVerifier: ReadOrAssert

4.5.2 A program exits sometimes.

When a program uses the `exit()` function to terminate itself, the outgoing flow of the block calling exit is 0. Unfortunately also for blocks that just call a function that may call the `exit()` function the incoming weight may be bigger than the outgoing weight. In this case the, the profile verifier checks if directly or via calls to other functions a call to `exit()` may be reached. If this is the case then the flow difference for this block is ignored.

```
void ProfileVerifierPass::recurseBasicBlock(BasicBlock *BB) {
  if (BBisVisited.find(BB) != BBisVisited.end()) return;

  DetailedBlockInfo DI;
  DI.BB = BB;
  DI.inWeight = DI.outWeight = 0;

  // Read predecessors.
  std::set<const BType*> ProcessedPreds;
  pred_const_iterator bpi = pred_begin(BB), bpe = pred_end(BB);
  // If there are none, check for (0,BB) edge.
  if (bpi == bpe) {
    DI.inWeight += ReadOrAssert(PI->getEdge(0,BB));
  }
  for (;bpi != bpe; ++bpi) {
    if (ProcessedPreds.insert(*bpi).second) {
      DI.inWeight += ReadOrAssert(PI->getEdge(*bpi,BB));
    }
  }

  // Read successors.
  std::set<const BType*> ProcessedSuccs;
  succ_const_iterator bbi = succ_begin(BB), bbe = succ_end(BB);
  double w = PI->getEdgeWeight(PI->getEdge(BB,0));
  if (w != ProfileInfo::MissingValue) {
    DI.outWeight += w;
  }
  for (;bbi != bbe; ++bbi) {
    if (ProcessedSuccs.insert(*bbi).second) {
      DI.outWeight += ReadOrAssert(PI->getEdge(BB,*bbi));
    }
  }

  DI.BBWeight = PI->getExecutionCount(BB);
  if (DI.BBWeight == ProfileInfo::MissingValue) {
    // Assertion message ...
  }
  if (DI.BBWeight < 0) {
    // Assertion message ...
  }

  if (DI.inWeight != DI.outWeight) {
    // Assertion message ...
  }
  if (DI.inWeight != DI.BBWeight) {
    // Assertion message ...
  }

  // Mark this block as visited, rescurse into successors.
  BBisVisited.insert(BB);
  for ( succ_const_iterator bbi = succ_begin(BB), bbe = succ_end(BB);
        bbi != bbe; ++bbi ) {
    recurseBasicBlock(*bbi);
  }
}
```

Figure 4.6: ProfileVerifier: recurseBasicBlock

Chapter 5

Results

5.1 Overview

Implementing an algorithm and verifying that it is working correctly is only part of the solution. Another important part is to actually verify that the algorithm is as efficient as promised. This chapter is all about testing the implementation in various ways and ensuring that it works as intended. Especially the runtime overhead was measured to check that the proposed increase in efficiency can indeed be observed.

Chapter 5.2 describes the methods used to do the correctness and performance measurements, Chapter 5.3 explains how the algorithms where tested to work correctly. Chapter 5.4 examines the performance penalty on the compiler during compile time and finally Chapter 5.5 shows and discusses the runtime performance results.

5.2 Used Methods

For most of the correctness and performance testing the SPEC CPU2000 [46] benchmark was used. This benchmark is a collection of C and C++ programs and a testing infrastructure that controls the whole process of building, running and verifying the benchmarked programs. SPEC CPU2000 tries to maximise the reproducibility of the performed tests so the tests in this thesis are not one-off results.

A special feature of the SPEC CPU2000 benchmark is that it is also possible to do a two phase build. In the first phase the program is build with instrumentation, then the program is executed with a training data set. In the second phase the program is rebuilt with the profiles generated by the training run to get the final executable.

The SPEC CPU2000 benchmark was used in five different build configurations to test various properties:

default A regular build.

profile A two phase build which used the old (naïve) edge profiling during the training run (each edge was instrumented). The final executable was non-instrumented.

benchmark_profile A regular build but with added naïve edge profiling. This final executable was instrumented.

opt_profile Same as **profile** but with the new optimal edge profiling.

benchmark_opt_profile Same as **benchmark_profile** but with optimal edge profiling.

Figure 5.1 gives the core configuration parameters for the five configurations, the **default** configuration was used as reference and the **benchmark_profile** and **benchmark_opt_profile** configurations were used to test the profiling overhead.

The `llvm-bytecode` script is a wrapper script around LLVM that generates LLVM intermediate representation from the C or C++ files, `llvm-linker` is a wrapper that performed linking, optional instrumentation and generated the final executable. Since the four last configurations are children of the first **default** configuration, they also use these wrapper scripts, but with different parameters.

Each executable was build by first executing `clang` on the C or C++ files and generate LLVM bytecode from those files. Then this bytecode was linked with `llvm-ld` together into one single big bytecode file. This bytecode file was then optimised with the `-std-compile-opts` from the LLVM `opt` tool. If requested (either because this is the first phase of a two phase build or for the runtime overhead testing) the resulting bytecode was instrumented and the profiling runtime library was linked into the bytecode. The `llc` was used to generate assembler code and finally `gcc` to compile the final executable from the assembler.

Setting up the SPEC CPU2000 benchmark and writing the scripts took some time, but this effort paid of when the final results had to be gathered and the benchmark was simply run to get all sorts of results.

5.2.1 Used LLVM Version

Since the LLVM code was constantly developed against the top of the development tree, the Revision 96682 of the LLVM source code repository from `http://llvm.org/svn/llvm-project/llvm/trunk` was used.

```
default=default=default=default:
CC  = llvm-bytecode $(LLVMOPT)
CXX = llvm-bytecode $(LLVMOPT) --gcc g++
CLD   = llvm-linker $(LLVMOPT)
CXXLD = llvm-linker $(LLVMOPT) --gcc g++
LLVMOPT = -d 2

default=default=profile:
PASS1_CFLAGS = -p 1 --profiler "-insert-edge-profiling"
PASS2_CFLAGS = -p 2

default=default=benchmark_profile:
LLVMOPT = -d 2 -p 1 --profiler "-insert-edge-profiling"

default=default=opt_profile:
PASS1_CFLAGS = -p 1 --profiler "-insert-optimal-edge-profiling"
PASS2_CFLAGS = -p 2

default=default=benchmark_opt_profile:
LLVMOPT = -d 2 -p 1 --profiler "-insert-optimal-edge-profiling"
```

Figure 5.1: Core part of the SPEC CPU2000 configuration

Also the `clang` project was already mature enough to be used for compiling and running the SPEC CPU2000 benchmark the `llvm-gcc` (which was used during most of the development) was removed and instead `clang` was used. Since LLVM and `clang` are developed together also the SVN Revision 96682 of the `clang` source code repository from `http://llvm.org/svn/llvm-project/cfe/trunk` was used.

5.2.2 Used Hardware

The tests, if not noted otherwise, were performed on a Dual Core AMD Opteron(tm) Processor 270 with 2 GHz CPU and 8 GB of RAM. The system was running Debian Linux with a 2.6.25-2-amd64 kernel.

5.3 Correctness

5.3.1 Profile Estimator

As already mentioned in Section 4.5 a LLVM pass was written that verified that a given profiling information was correct by testing the flow condition in each basic block of the program. The flow condition states that the sum of the

execution counts of the incoming edges has to be equal to the execution count of the block itself which in turn has to be equal to the sum of the execution counts of the outgoing edges.

Some small programs where used to verify that the profile estimator does everything it should, namely splitting the flow at branches and incrementing the flow inside a loop. Then the estimator was used to estimate all of the C and C++ programs in the SPEC CPU2000 benchmark and the verifier (Section 4.5) was used to check that the flow condition was nowhere violated.

This verification showed numerous bugs and deficiencies at first (Sections 4.2 and 4.3) but after these problems where solved the profile estimator is believed to work properly on all CFGs (or to gracefully bail out in case the CFG is really not determinable with this algorithm).

5.3.2 Instrumentation Algorithm and Profiling Framework

The instrumentation algorithm was checked in three ways:

- For four small C programs the whole process of estimating the CFG, calculating the MST and instrumenting the graph was observed and verified.

- Then profiles for these programs were generated by executing the programs this was also done with the naïve old profile implementation. The profiles from the old implementation (which instruments all edges and is thus assumed to be correct...) and from the new implementation were compared, also the verifier was used to check both profiles.

- When those small programs worked properly and the profiles where the same for the old and new implementation, this process was iterated on all the C and C++ programs of the SPEC CPU2000 benchmark and again the profiles where compared.
  Except for the cases where the program terminated because of a call to `exit` the profiles were exactly the same.

Since the profiling and the comparing of the profiles was done with the tools LLVM provided for this purpose, this also tested all the infrastructure in the LLVM, namely the profile loader and the `llvm-prof` tool. Some bugs were encountered, most notably that (with the old implementation) no edge instrumentation was done in functions with only one basic block.



Figure 5.2: Percentage of instrumented edges.

5.4 Results Compile Time

One of the goals was to reduce the number of edges needed to generate the execution profile of a program. That the generated profiles were indeed correct was discussed in the previous section, this section focusses on the number of edges inserted. Figure 5.2 shows a quick overview on the percentage of instrumented edges and Table 5.1 shows the detailed figures for each program in the SPEC CPU2000 benchmark.

On average, the naïve algorithm instrumented **93.5%** of the edges and the optimal algorithm instrumented **47.5%**. That is an improvement of almost a **factor of 2**.

Is is interesting to note that even the naïve implementation does not instrument 100% of the edges, since it did not instrument the exiting edges $(v, 0)$ that the optimal profiling also considered. For the comparisons to be fair this small "optimisation" of the naïve algorithm has to be taken into account.

Since the naïve instrumentation does not instrument exiting edges the difference in the number of instrumented edges for small functions was not as big. This raised the question whether the naïve algorithm works as well as the optimal algorithm for programs with many small functions, this is covered in the next section.

Program	Edges Instrumented[%]	
	Naïve	Optimal
164.gzip	91.907	49.461
175.vpr	88.993	37.577
176.gcc	94.160	43.161
181.mcf	93.811	39.902
186.crafty	95.420	38.543
197.parser	94.028	40.577
252.eon	87.879	43.003
253.perlbmk	93.587	56.435
254.gap	92.799	54.290
255.vortex	97.774	58.094
256.bzip2	90.581	61.605
300.twolf	94.637	39.289
177.mesa	86.687	40.817
179.art	93.927	38.630
183.equake	87.845	37.937
188.ammp	94.817	38.867
Average	93.491	47.498

Table 5.1: Percentage of instrumented edges.

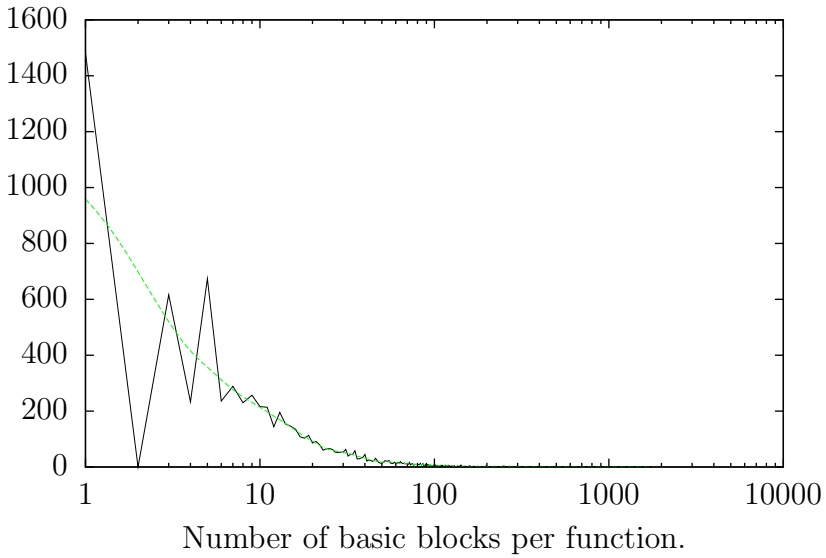


Figure 5.3: Number of Functions with a given Number of Blocks

5.4.1 Profiling Small Functions

Figure 5.3 shows the distribution of function sizes (measured in the number of basic blocks) in the SPEC CPU2000 benchmark. As expected there is a huge amount of functions that have only 1 or 3 blocks, since unconditional branches are optimised there are no functions with two blocks. The median of the function size is at 9 basic blocks, of the 7770 functions in the benchmark, 4016 have 9 basic blocks or less. There are only 874 functions with 50 basic blocks or more, but also 2 functions with more than 1000 basic blocks.

Figure 5.4 shows how the naïve and optimal number of instrumented edges is distributed over the function size. For functions with two to six basic blocks the optimal number of instrumented edges is around 42%, for functions with more than eleven blocks the optimal number of instrumented edges is around 47% percent. As with the difference in the number of instrumented edges per program (see Section 5.4.2) the differences in instrumentation between small and big functions can be attributed to the fact that small functions have less cycles, thus less edges are instrumented.

For functions of size 1 both algorithms perform equally well because even the naïve algorithm does not instrument both the incoming and the outgoing edge. The number of instrumented edges for functions of size 3 and 4 is already considerably larger for the naïve implementation and grows from there on with the functions size. The number of instrumented edges for the naïve algorithm never reaches 100% since the number of basic blocks exiting the function and thus the number of basic blocks that have no virtual edge attached also grows.

Although the naïve algorithm does not instrument all edges it is still far away

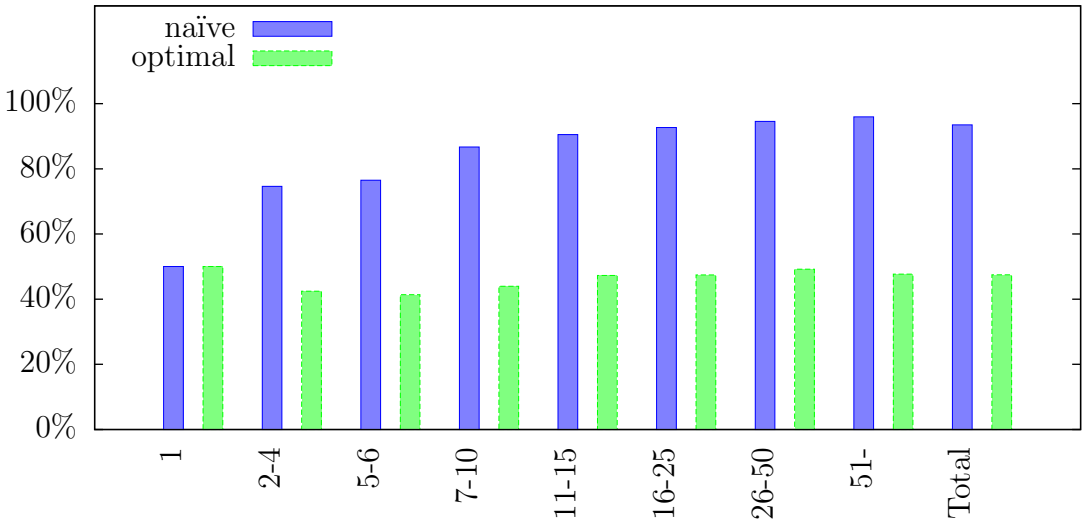


Figure 5.4: Instrumented edges per function size.

from being optimal since most functions have 3 or more blocks and for this functions the optimum is usually below 50%.

5.4.2 Differences in Optima

One interesting thing to note is that for some of the benchmark programs the optimal instrumentation uses less edges than for others, Table 5.1 shows the programs sorted by the amount of optimal edges. Especially the `256.bzip2` program, but also `255.vortex` and `253.perlbmk` are far above the average, although there are also functions below the average this is not as pronounced.

As discussed in Section 2.4.4 the number of instrumented edges in a function is the number of all edges minus the number of basic blocks. This leads to the conclusion that in programs where more edges are instrumented the blocks in the functions are more tightly interconnected. Also since the optimal instrumentation can be seen as breaking up each cycle in a CFG by instrumenting at least one edge on this cycle (see Section 2.4.5) it can be assumed that programs where more edges are instrumented have more cycles in their CFGs and the other programs are more linear in nature.

5.4.3 Build Times

An important question when changing a compiler is: how does the changes affect compile time? The SPEC CPU2000 benchmark has no native way of accurately measuring the build times, so the standard UNIX tool `time` was

Program	Edges Instrumented[%]	
	Naïve	Optimal
175.vpr	88.993	37.577
183.equake	87.845	37.937
186.crafty	95.420	38.543
179.art	93.927	38.630
188.ammp	94.817	38.867
300.twolf	94.637	39.289
181.mcf	93.811	39.902
197.parser	94.028	40.577
177.mesa	86.687	40.817
252.eon	87.879	43.003
176.gcc	94.160	43.161
164.gzip	91.907	49.461
254.gap	92.799	54.290
253.perlbmk	93.587	56.435
255.vortex	97.774	58.094
256.bzip2	90.581	61.605
Maximum	97.774	61.605
Minimum	86.687	37.577
Distance Min-Max	11.087	24.028

Table 5.2: Percentage of instrumented edges (sorted by Optimal).

used. The build was done five times and for each program the fastest values were chosen, Table 5.3 shows the resulting build times in seconds for:

None Best build time without any instrumentation.

Naïve Best build time with naïve instrumentation.

Optimal Best build time with optimal instrumentation.

Naïve Overhead The build time overhead of the naïve instrumentation as compared to no instrumentation in percent.

Optimal Overhead As **Naïve Overhead** but for the optimal instrumentation.

Improvement Improvement of the overhead between naïve and optimal instrumentation, lower values mean more improvement.

The build time with naïve instrumentation is on average about 30% higher than the build time without performing any instrumentation, the build time overhead with optimal instrumentation is 19%. Although the optimal algorithm is more complicated than the naïve one (it has to do an estimation, calculate a maximum spanning tree, insert counter initialisation code and insert the counters itself) the faster compile times result from the fact that less counters are inserted and thus the intermediate representation has to be modified less often.

5.5 Results Run Time

5.5.1 Runtime Results amd64 Hardware

The runtime results where obtained by running the uninstrumented, the naïvely and the optimally instrumented programs from the SPEC CPU2000 benchmark five times. During these five runs the runtimes itself changed considerably, e.g. for the `181.mcf` program the longest run used 31.1% more time than the fastest run, on average the longest ran was 3.7% longer than the fastest run. Because of the varying runtimes Section 5.5.2 compares the amd64 results with values from a x86_64 platform. To verify this results on a different architecture Section 5.5.3 presents runtime results from a ppc32 platform.

Table 5.4 shows the fastest runs for each program on an amd64 platform, the columns are as follows:

None Best runtime without any instrumentation.

Naïve Best runtime with the naïve instrumentation.

Optimal Best runtime with optimal instrumentation.

Program	Instrumentation[s]			Overhead[%]		Improvement
	None	Naïve	Optimal	Naïve	Optimal	
164.gzip	2.41	2.91	2.78	20.75	15.35	74.00
175.vpr	6.21	7.94	7.30	27.86	17.55	63.01
176.gcc	56.23	79.07	70.14	40.62	24.74	60.90
177.mesa	29.10	50.30	40.75	72.85	40.03	54.95
179.art	1.28	1.47	1.42	14.84	10.94	73.68
181.mcf	1.66	1.90	1.86	14.46	12.05	83.33
183.equake	1.43	1.62	1.58	13.29	10.49	78.95
186.crafty	9.55	11.85	10.97	24.08	14.87	61.74
188.ammp	9.01	13.28	11.20	47.39	24.31	51.29
197.parser	5.08	6.63	6.13	30.51	20.67	67.74
252.eon	74.73	86.41	82.91	15.63	10.95	70.03
253.perlbmk	23.61	33.11	29.19	40.24	23.63	58.74
254.gap	17.42	23.12	21.07	32.72	20.95	64.04
255.vortex	19.99	28.73	25.11	43.72	25.61	58.58
256.bzip2	1.95	2.32	2.20	18.97	12.82	67.57
300.twolf	10.90	13.99	12.92	28.35	18.53	65.37
Average				30.39	18.97	62.41
Minimum				13.29	10.49	
Maximum				72.85	40.03	

Table 5.3: Build times from the SPEC CPU2000 benchmark.

Naïve Overhead The runtime overhead of the naïve instrumentation as compared to no instrumentation in percent.

Optimal Overhead As **Naïve Overhead** but for the optimal instrumentation.

Improvement Improvement of the overhead between naïve and optimal instrumentation, lower values mean more improvement.

The average runtime overhead on the SPEC CPU2000 benchmark for the naïve profiling is **14.4%** and **7.12%** for the optimal profiling which is a reduction of the overhead by slightly more than 50%, but there are some more extreme values. For example the optimal overhead for the `181.mcf` program was only 24% of the naïve overhead, so the overhead was reduced by 3/4. There are also examples where the improvement was almost non existent, for the `256.bzip2` program the overhead was only 15% lower for the optimal version.

The differences in the overhead improvements could not be attributed to any definitive factors, the number of instrumented edges certainly did not make a difference: the `256.bzip2` and `255.vortex` programs had the highest percentage of instrumented edges, yet the `256.bzip2` program runtime overhead was only improved by 15% but the overhead for the `255.vortex` program was improved by 45%.

5.5.2 Runtime Results x86_64 Hardware

Since the amd64 hardware delivered runtimes that where not as stable as desired, a different hardware platform was used to confirm the results. For the following results a 8 Core Intel(R) Xeon(R) CPU with 3 GHz and 24GB of RAM running Debian Linux with a 2.6.30-perfctr kernel was used, on this hardware the runtimes differed by at most 3.7% and by 0.6% on average.

Table 5.5 shows the fastest runs for each program on and x86_64 platform, the columns are the same as in Section 5.5.1. The x86_64 platform shows approximately the same improvements as the amd64 platform.

5.5.3 Runtime Results ppc32 Hardware

To verify the results from Section 5.5.1 on a non-x86 architecture the SPEC CPU2000 benchmark was also performed on a 1.4 GHz iBook G4 with a 32-bit PPC processor and 1.5 GB of RAM running a fully patched version of Mac OS 10.5. The system was in single user console mode, meaning that only a bare shell with no GUI was active, this reduced the variation of runtimes to be almost unnoticeable.

Program	Instrumentation[s]			Overhead[%]		Improvement
	None	Naïve	Optimal	Naïve	Optimal	
164.gzip	142	175	156	23.51	9.86	41.94
175.vpr	159	173	164	8.58	2.60	30.31
176.gcc	97	118	110	21.10	13.06	61.92
181.mcf	309	330	314	6.64	1.58	23.87
186.crafty	55	70	61	27.03	10.81	39.98
197.parser	237	266	249	12.15	5.00	41.12
252.eon	63	67	66	6.41	5.29	82.58
253.perlbmk	151	186	166	22.77	9.76	42.87
254.gap	107	117	112	9.69	4.44	45.82
255.vortex	120	162	142	35.90	18.49	51.50
256.bzip2	160	188	184	17.82	15.11	84.79
300.twolf	272	293	288	7.42	5.58	75.27
177.mesa	110	113	105	2.77	-4.09	-147.58
179.art	251	257	255	2.72	1.68	61.85
183.equake	118	125	124	6.01	4.81	80.01
188.ammp	202	243	222	19.95	10.00	50.11
Average				14.40	7.12	49.46
Minimum				2.72	-4.09	
Maximum				35.9	18.49	

Table 5.4: Best runtimes for each SPEC CPU2000 program (amd64).

Program	Instrumentation[s]			Overhead[%]		Improvement
	None	Naïve	Optimal	Naïve	Optimal	
164.gzip	88	106	96	20.32	8.36	41.13
175.vpr	70	79	72	12.25	3.31	26.99
176.gcc	55	67	63	21.76	14.15	65.01
181.mcf	78	88	82	12.62	4.89	38.78
186.crafty	30	38	32	26.32	6.44	24.47
197.parser	113	131	122	16.52	8.62	52.17
252.eon	35	40	37	13.28	5.00	37.67
253.perlbmk	66	93	75	40.72	13.49	33.14
254.gap	48	59	55	23.24	14.50	62.38
255.vortex	58	81	68	39.44	16.73	42.41
256.bzip2	71	92	84	29.30	18.11	61.80
300.twolf	95	103	98	8.74	3.48	39.83
177.mesa	52	58	54	13.10	5.69	43.48
179.art	35	42	40	19.80	14.51	73.28
183.equake	70	73	71	5.08	1.78	35.08
188.ammp	91	106	99	16.90	9.77	57.79
Average				19.96	9.30	46.60
Minimum				5.08	1.78	
Maximum				40.72	18.11	

Table 5.5: Best runtimes for each SPEC CPU2000 program (x86_64).

Program	Instrumentation[s]			Overhead[%]		Improvement
	None	Naïve	Optimal	Naïve	Optimal	
164.gzip	307	431	373	40.34	21.26	52.71
175.vpr	478	540	504	13.01	5.43	41.76
181.mcf	852	877	866	2.96	1.63	54.92
256.bzip2	455	554	533	21.83	17.05	78.09
300.twolf	790	929	866	17.69	9.65	54.57
177.mesa	288	348	318	20.72	10.47	50.54
179.art	1169	1310	1298	12.12	11.02	90.91
183.equake	522	538	533	3.00	2.20	73.38
Average				16.46	9.84	59.78
Minimum				2.96	1.63	
Maximum				40.34	21.26	

Table 5.6: Best runtimes for selected SPEC CPU2000 program (ppc32).

Unfortunately some of the programs did not build and some of the programs did not run properly since the ppc32 target is not especially well supported by LLVM but no modifications were made to either LLVM or the SPEC CPU2000 benchmark to make the programs run. This ensured that the results could be compared without bias. Also, because of the low power of the system, only 3 runs were performed to keep the testing time down.

The results show the same trend as the amd64 results, namely that the runtime overhead for the naïve instrumentation is approx. 16% and the optimal overhead is approx. 10%. This improvement is slightly less than on amd64 hardware but since the overall characteristics of the runtimes are quite different between the x86 and ppc32 architectures the cause of this difference was not determined. Still the results suggest that approx. 50% in runtime improvement can be expected on different hardware platforms too.

5.5.4 Effectiveness of Profile Estimator

The profile estimator is supposedly placing the edge counters where they are less likely to be executed, thus reducing the runtime overhead of the program. To verify this, the instrumentation overhead of a version with active profile estimator was compared to a version where the estimation was not used and the maximum spanning tree uses a random edges sorting. Since the difference between this two variants was small the x86_64 platform (as in Section 5.5.2) was used for the tests because it delivers more accurate results.

Program	Instrumentation[s]			Overhead[%]		Difference
	None	Estimation	Random	Estimation	Random	
164.gzip	88	95	97	8.78	10.25	116.81
175.vpr	70	72	73	3.74	4.86	130.01
176.gcc	55	62	61	13.81	10.78	78.05
181.mcf	79	82	86	4.09	9.57	233.62
186.crafty	30	32	32	6.70	8.62	128.57
197.parser	113	122	123	8.17	8.57	104.91
252.eon	35	37	38	4.82	8.94	185.64
253.perlbmk	66	75	82	13.46	23.62	175.42
254.gap	48	55	54	14.14	13.55	95.79
255.vortex	58	68	70	16.42	18.96	115.44
256.bzip2	71	84	86	18.23	20.71	113.59
300.twolf	95	98	100	3.36	5.87	174.40
177.mesa	52	54	55	5.65	6.37	112.75
179.art	35	40	39	13.54	12.28	90.65
183.equake	71	71	72	-0.88	1.31	-149.92
188.ammp	91	99	99	9.69	9.74	100.54
Average				8.98	10.87	121.04
Minimum				-0.88	1.31	
Maximum				18.23	23.62	

Table 5.7: Runtimes with and without Profile Estimator

In Figure 5.7 the results are shown, the profiling overhead is approx. 20% higher with random edge placement. For some the `mcf.181` program the random variant is 133% slower than the estimated variant, but there are also programs that are slightly faster with random placement.

5.5.5 Using Profiling Data

One of the only ways to use the recorded profiling data for profile based optimisations was to use it instead of the estimated edge weights for calculating the maximum spanning tree when instrumenting the program. (LLVM currently has no other profile based optimisations implemented.)

In Figure 5.8 the results are shown, the profiling overhead was about 7% less for the edges placed with real profiling data as opposed to the edges placed by the estimated data. It is assumed that the improvement is only small because of the fact that most of the work of a program happens inside of loops and

Program	Instrumentation[s]			Overhead[%]		Difference
	None	Estimation	Prof. Data	Estimation	Prof. Data	
164.gzip	87.7	95.6	96.5	8.97	10.03	111.83
175.vpr	69.5	72.3	72.3	4.07	4.02	98.85
176.gcc	54.6	62.1	60.7	13.83	11.19	80.95
181.mcf	78.3	81.8	81.1	4.47	3.63	81.15
186.crafty	29.8	31.8	31.5	6.50	5.67	87.28
197.parser	113.2	122.3	121.0	7.99	6.88	86.12
252.eon	35.0	36.7	35.0	4.92	0.05	1.02
253.perlbmk	66.4	75.0	80.5	12.93	21.17	163.68
254.gap	47.8	54.6	52.4	14.01	9.55	68.11
255.vortex	58.4	68.2	66.4	16.74	13.62	81.34
256.bzip2	70.9	84.5	83.8	19.23	18.21	94.68
300.twolf	95.0	98.1	99.0	3.25	4.19	128.91
177.mesa	51.6	54.5	53.9	5.71	4.51	78.96
179.art	34.9	39.9	39.9	14.39	14.50	100.78
183.equake	71.4	70.8	71.0	-0.83	-0.67	81.73
188.ammp	90.5	99.4	99.5	9.80	9.87	100.62
Average				9.12	8.52	93.43
Minimum				-0.83	-0.67	
Maximum				19.23	21.17	

Table 5.8: Runtimes with Estimator and Profiling Data

these loops have to be instrumented in both variants with the same amount of counters.

Chapter 6

Conclusions

Being able to profile a program and use this information to improve the program (either manually or during the recompilation) is a valuable addition to the development process. Doing the necessary instrumentation efficiently (during both compile time and runtime) is necessary to ensure continued use of these techniques.

Implementing Knuth's algorithm for optimal edge counter placement resulted in 50% less instrumented edges, a 35% drop in compile time overhead and a 45% drop in runtime overhead. The results were comparable on different hardware platforms (amd64, x86_64, ppc32) which promises similar results for other architectures.

Trying to place the instrumented edges in less often executed regions of the program by using a naïve profiling estimator (see Sections 2.3.1 and 2.4.1) was also successful: doing the optimal instrumentation with a random spanning tree (instead of the maximum spanning tree that used the estimates) increased the profiling overhead by 20%. So the profile estimator is not a huge factor in improving the profiling runtime overhead, but it contributes around 8% to the 45% overall drop in runtime overhead.

Preparing the profiling information to be used in the backend was very time consuming. For every pass that transforms the CFG an equivalent transformation has to be done on the profiling information. The process of adding all this transformations for the profiling information involved finding the passes that modify the CFG, finding out where they do the modifications and what those modifications are and finally adding the same transformation on the profiling information.

Unfortunately there were no profile based optimisations realised in LLVM so the effects of having the profiling information available could only be validated against the profiling instrumentation itself (see Section 5.5.5). The achieved 7% overhead reduction is only small but several profile based optimisations together could substantially improve the program.

6.1 Future Work

Since the profiling data driven edge placement is not much better than the one with estimated profiles the runtime overhead of the profiling framework can not be improved too much any more but instrumenting the code even less and estimating parts of the result may further reduce the overhead.

Also a timed approach could be considered where the counter update code is switched on/off during runtime e.g. the counters are only updated during 10% of the running time of the program to deliver approximate results.

Finally a lot of future work can be done by implementing profile based optimisations in LLVM and by really using the power these profiling tools give to the programmer.

Acknowledgements

My thanks go to Dietmar Ebner for introducing me to the topics of this thesis and to Andreas Krall for supervising this thesis at the Vienna University of Technology. I also want to thank the Institute of Computer Languages and the Compilers and Languages Group for letting me use their spare workplace, this was tremendously helpful for keeping in contact with Dietmar and Andreas.

The participants on the LLVM mailing lists were extremely helpful and provided advice and guidance on the implementation issues and technical details of the LLVM compiler—thank you for that.

I want to thank the people that I met during my studies: Alexandra Schuster and Tamara Wenzel who became true friends and Stefan Rümmele for being the academic I never will be.

I want to thank my dear friends Doris Fried, Rafael Hartmann and Klaus Böswart for never thinking that my studies are a bad idea. Special thanks to my family: my parents Franz and Maria Neustifter and my brothers and sisters and their families for their moral support.

And most importantly: Thank you Eva for everything.

Literature

[1] A. V. Aho, S. C. Johnson, and J. D. Ullman. Code generation for expressions with common subexpressions. *J. ACM*, 24(1):146–160, 1977.

[2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, US ed edition, 1986.

[3] Alfred V. Aho and Jeffrey D. Ullman. *The Theory of Parsing, Translation, and Compiling*. Prentice Hall Professional Technical Reference, 1972.

[4] Alfred V. Aho and Jeffrey D. Ullman. *Principles of Compiler Design*. Addison-Wesley, August 1977.

[5] F. E. Allen. Program optimization. *Annual Review in Automatic Programming*, 5, 1969.

[6] F. E. Allen and J. Cocke. A program data flow analysis procedure. *Commun. ACM*, 19(3):137, 1976.

[7] Frances E. Allen. Control flow analysis. *SIGPLAN Not.*, 5(7):1–19, 1970.

[8] Andrei Alvares. Static profile patch. http://lists.cs.uiuc.edu/pipermail/llvm-commits/Week-of-Mon-20090907/086955.html, 2009.

[9] Glenn Ammons, Thomas Ball, and James R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. In *Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation*, pages 85–96, Las Vegas, Nevada, United States, 1997. ACM.

[10] Jennifer M. Anderson, Lance M. Berc, Jeffrey Dean, Sanjay Ghemawat, Monika R. Henzinger, Shun-Tak A. Leung, Richard L. Sites, Mark T. Vandevoorde, Carl A. Waldspurger, and William E. Weihl. Continuous profiling: where have all the cycles gone? *SIGOPS Oper. Syst. Rev.*, 31(5):1–14, 1997.

[11] Thomas E. Anderson and Edward D. Lazowska. Quartz: a tool for tuning parallel program performance. *SIGMETRICS Perform. Eval. Rev.*, 18(1):115–125, 1990.

[12] C. T. Apple. The program monitor, a device for program performance measurement. In *Proceedings of the 1965 20th national conference*, pages 66–75, Cleveland, Ohio, United States, 1965. ACM.

[13] Ziya Aral and Ilya Gertner. Non-intrusive and interactive profiling in parasight. In *Proceedings of the ACM/SIGPLAN conference on Parallel programming: experience with applications, languages and systems*, pages 21–30, New Haven, Connecticut, United States, 1988. ACM.

[14] Stefan Arnborg. A note on the assignment of measurement points for frequency counts in structured programs. *BIT Numerical Mathematics*, 14(3):273–278, 1974.

[15] Matthew Arnold, Michael Hind, and Barbara Ryder. An empirical study of selective optimization. In *Languages and Compilers for Parallel Computing*, pages 49–67. Springer Berlin / Heidelberg, 2001.

[16] Matthew Arnold and Barbara G. Ryder. A framework for reducing the cost of instrumented code. In *Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 168–179, Snowbird, Utah, United States, 2001. ACM.

[17] Thomas Ball. Efficiently counting program events with support for on-line queries. *ACM Trans. Program. Lang. Syst.*, 16(5):1399–1410, 1994.

[18] Thomas Ball and James R. Larus. Branch prediction for free. *SIGPLAN Not.*, 28(6):300–313, 1993.

[19] Thomas Ball and James R. Larus. Optimally profiling and tracing programs. *ACM Trans. Program. Lang. Syst.*, 16(4):1319–1360, 1994.

[20] Thomas Ball and James R. Larus. Efficient path profiling. In *Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*, pages 46–57, Paris, France, 1996. IEEE Computer Society.

[21] Thomas Ball, Peter Mataga, and Mooly Sagiv. Edge profiling versus path profiling: the showdown. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 134–148, San Diego, California, United States, 1998. ACM.

[22] Incorporated. Bell Telephone Laboratories, Lucent Technologies Inc., and AT&T Corporation. *Unix Seventh Edition Manual*. Bell Labs, 1979.

[23] Jon Louis Bentley. Writing efficient code. Technical report, Department of Computer Science at Carnegie Mellon University, Pittsburg, April 1981.

[24] Rolf Berrendorf, Heinz Ziegler, and Bernd Mohr. PCL - the performance counter library. http://www.fz-juelich.de/jsc/PCL/.

[25] S. Browne, J. Dongarra, N. Garner, K. London, and P. Mucci. A scalable cross-platform infrastructure for application performance tuning us-

ing hardware counters. In *Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, page 42, Dallas, Texas, United States, 2000. IEEE Computer Society.

[26] Robert G. Burger and R. Kent Dybvig. An infrastructure for Profile-Driven dynamic recompilation. In *Computer Languages, International Conference on*, volume 0, page 240, Los Alamitos, CA, USA, 1998. IEEE Computer Society.

[27] Brad Calder, Peter Feller, and Alan Eustace. Value profiling. In *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, pages 259–269, Research Triangle Park, North Carolina, United States, 1997. IEEE Computer Society.

[28] Peter Calingaert. System performance evaluation: survey and appraisal. *Commun. ACM*, 10(1):12–18, 1967.

[29] V. G Cerf. Measurement of recursive programs. Technical report, Los Angeles School of Engineering and Applied Science, May 1970.

[30] Pohua P. Chang, Scott A. Mahlke, William Y. Chen, and Wen mei W. Hwu. Profile-guided automatic inline expansion for c programs. *Softw. Pract. Exper.*, 22(5):349–369, 1992.

[31] Pohua P. Chang, Scott A. Mahlke, and Wen mei W. Hwu. Using profile information to assist classic code optimizations. *Softw. Pract. Exper.*, 21(12):1301–1321, 1991.

[32] John Cocke. *Programming languages and their compilers: Preliminary notes*. Courant Institute of Mathematical Sciences, New York University, 1969.

[33] John Cocke. Global common subexpression elimination. *SIGPLAN Not.*, 5(7):20–24, 1970.

[34] John Cocke and Raymond Miller. Some analysis techniques for optimizing computer programs. In *Proc. Second Intl. Conf. of Systems Sciences*, Hawaii, 1969.

[35] Wikipedia contributors. Low level virtual machine. http://en.wikipedia.org/wiki/Low_Level_Virtual_Machine, 2010.

[36] Keith D. Cooper, Alexander Grosul, Timothy J. Harvey, Steven Reeves, Devika Subramanian, Linda Torczon, and Todd Waterman. ACME: adaptive compilation made efficient. In *Proceedings of the 2005 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 69–77, Chicago, Illinois, USA, 2005. ACM.

[37] C.A. Coutant, R.E. Griswold, and D.R. Hanson. Measuring the performance and behavior of icon programs. *IEEE Transactions on Software Engineering*, 9(1):93–103, 1983.

[38] Saumya Debray and William Evans. Profile-guided code compression. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 95–105, Berlin, Germany, 2002. ACM.

[39] J.L. Elshoff. The influence of structured programming on PL/I program profiles. *Software Engineering, IEEE Transactions on*, SE-3(5):364–368, 1977.

[40] Stéphane Eranian. perfmon2 - the hardware-based performance monitoring interface for linux. http://perfmon2.sourceforge.net/, 2002.

[41] Joseph A. Fisher and Stefan M. Freudenberger. Predicting conditional branch directions from previous runs of a program. *SIGPLAN Not.*, 27(9):85–95, 1992.

[42] Ira R. Forman. On the time overhead of counters and traversal markers. In *Proceedings of the 5th international conference on Software engineering*, pages 164–169, San Diego, California, United States, 1981. IEEE Press.

[43] Aaron J. Goldberg and John L. Hennessy. Performance debugging shared memory multiprocessor programs with MTOOL. In *Proceedings of the 1991 ACM/IEEE conference on Supercomputing*, pages 481–490, Albuquerque, New Mexico, United States, 1991. ACM.

[44] Susan L. Graham, Peter B. Kessler, and Marshall K. Mckusick. Gprof: A call graph execution profiler. In *Proceedings of the 1982 SIGPLAN symposium on Compiler construction*, pages 120–126, Boston, Massachusetts, United States, 1982. ACM.

[45] Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. An execution profiler for modular programs. *Software: Practice and Experience*, 13(8):671–685, 1983.

[46] John L. Henning. SPEC CPU2000: measuring CPU performance in the new millennium. *Computer*, 33(7):28–35, 2000.

[47] Donald J. Herman and Fred C. Ihrer. The use of a computer to evaluate computers. In *Proceedings of the April 21-23, 1964, spring joint computer conference*, pages 383–395, Washington, D.C., 1964. ACM.

[48] MIPS Computer Systems Inc. *Language Programmer's Guide*. MIPS Computer Systems, California, 1986.

[49] Daniel H. H. Ingalls. FETE: a fortran execution time estimator. Technical report, Stanford University, 1971.

[50] Marty Itzkowitz, Brian J. N. Wylie, Christopher Aoki, and Nicolai Kosche. Memory profiling using hardware counters. In *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, page 17. IEEE Computer Society, 2003.

[51] Lawrence J. Kenah, Ruth E. Goldenberg, and Simon F. Bate. *Vax/Vms Internals and Data Structures: Version 4.4*. Digital Press, 4th revised edition edition, May 1988.

[52] Donald E. Knuth. An empirical study of FORTRAN programs. *Software: Practice and Experience*, 1(2):105–133, 1971.

[53] Donald E. Knuth and Francis R. Stevenson. Optimal measurement points for program frequency counts. *BIT Numerical Mathematics*, 13(3):313–322, 1973.

[54] Nick Kufrin. PerfSuite: an accessible, open source performance analysis environment for linux. In *Proceedings of the 6th International Conference on Linux Clusters: The HPC Revolution*, Chapel Hill, NC, April 2005.

[55] J. R. Larus. Abstract execution: a technique for efficiently tracing programs. *Softw. Pract. Exper.*, 20(12):1241–1258, 1990.

[56] James R Larus and Thomas Ball. Rewriting executable files to measure program behavior. *SOFTWARE PRACTICE & EXPERIENCE*, 24:197—218, 1994.

[57] J.R. Larus. Efficient program tracing. *Computer*, 26(5):52–61, 1993.

[58] Edward S. Lowry and C. W. Medlock. Object code optimization. *Commun. ACM*, 12(1):13–22, 1969.

[59] J. Nievergelt. On the automatic simplification of computer programs. *Commun. ACM*, 8(6):366–370, 1965.

[60] Mikael Pettersson. Linux performance counters driver. http://perfctr.sourceforge.net/.

[61] Karl Pettis and Robert C. Hansen. Profile guided code positioning. *SIGPLAN Not.*, 25(6):16–27, 1990.

[62] R.L. Probert. Optimal insertion of software probes in Well-Delimited programs. *IEEE Transactions on Software Engineering*, 8(1):34–42, 1982.

[63] Reese T. Prosser. Applications of boolean matrices to the analysis of flow diagrams. In *Papers presented at the December 1-3, 1959, eastern joint IRE-AIEE-ACM computer conference*, pages 133–138, Boston, Massachusetts, 1959. ACM.

[64] Thomas Reps, Thomas Ball, Manuvir Das, and James Larus. The use of program profiling for software maintenance with applications to the year 2000 problem. In *Proceedings of the 6th European SOFTWARE ENGINEERING conference held jointly with the 5th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 432–449, Zurich, Switzerland, 1997. Springer-Verlag New York, Inc.

[65] Edward C. Russell. Automatic program analysis. Technical report, Los Angeles School of Engineering and Applied Science, March 1969.

[66] Alan Dain Samples. *Profile-driven compilation.* PhD thesis, University of California at Berkeley, 1992.

[67] Edwin Hallowell Satterthwaite. *Source language debugging tools.* PhD thesis, Stanford University, 1975.

[68] Tim A. Wagner, Vance Maverick, Susan L. Graham, and Michael A. Harrison. Accurate static estimators for program optimization. In *Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 85–96, Orlando, Florida, United States, 1994. ACM.

[69] David W. Wall. Global register allocation at link time. *SIGPLAN Not.*, 21(7):264–275, 1986.

[70] David W. Wall. Predicting program behavior using real or estimated profiles. *SIGPLAN Not.*, 26(6):59–70, 1991.

[71] Youfeng Wu and James R. Larus. Static branch frequency and program profile analysis. In *Proceedings of the 27th annual international symposium on Microarchitecture*, pages 1–11, San Jose, California, United States, 1994. ACM.

[72] A. P. Yershov. ALPHA–an automatic programming system of high efficiency. *Journal of the ACM*, 13(1):17–24, 1966.

[73] Cliff Young and Michael D. Smith. Improving the accuracy of static branch prediction using branch correlation. In *Proceedings of the sixth international conference on Architectural support for programming languages and operating systems*, pages 232–241, San Jose, California, United States, 1994. ACM.

[74] Marco Zagha, Brond Larson, Steve Turner, and Marty Itzkowitz. Performance analysis using the MIPS r10000 performance counters. In *Proceedings of the 1996 ACM/IEEE conference on Supercomputing (CDROM)*, page 16, Pittsburgh, Pennsylvania, United States, 1996. IEEE Computer Society.