

RICE UNIVERSITY

Low-Level Haskell Code: Measurements and Optimization Techniques

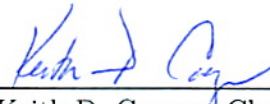
by

David McCarthy Peixotto

A THESIS SUBMITTED
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE

Doctor of Philosophy

APPROVED, THESIS COMMITTEE:



Keith D. Cooper, Chair
L. John and Ann H. Doerr Professor
Computer Science



Vivek Sarkar
E.D. Butcher Professor
Computer Science



Wotao Yin
Assistant Professor
Computational and Applied Mathematics

HOUSTON, TEXAS
MAY 2012

Abstract

Low-Level Haskell Code: Measurements and Optimization Techniques

by

David M. Peixotto

Haskell is a lazy functional language with a strong static type system and excellent support for parallel programming. The language features of Haskell make it easier to write correct and maintainable programs, but execution speed often suffers from the high levels of abstraction. While much past research focuses on high-level optimizations that take advantage of the functional properties of Haskell, relatively little attention has been paid to the optimization opportunities in the low-level imperative code generated during translation to machine code. One problem with current low-level optimizations is that their effectiveness is limited by the obscured control flow caused by Haskell's high-level abstractions. My thesis is that trace-based optimization techniques can be used to improve the effectiveness of low-level optimizations for Haskell programs. I claim three unique contributions in this work.

The first contribution is to expose some properties of low-level Haskell codes by looking at the mix of operations performed by the selected benchmark codes and comparing them to the low-level codes coming from traditional programming languages. The low-level measurements reveal that the control flow is obscured by indirect jumps caused by the implementation of lazy evaluation, higher-order functions, and the separately managed stacks used by Haskell programs.

My second contribution is a study on the effectiveness of a dynamic binary trace-based optimizer running on Haskell programs. My results show that while viable program traces frequently occur in Haskell programs the overhead associated with maintaining the traces in a dynamic optimization system outweigh the benefits we get from running the traces. To reduce the runtime overheads, I explore a way to find traces in a separate profiling step.

My final contribution is to build and evaluate a static trace-based optimizer for Haskell programs. The static optimizer uses profiling data to find traces in a Haskell program and then restructures the code around the traces to increase the scope available to the low-level optimizer. My results show that we can successfully build traces in Haskell programs, and the optimized code yields a speedup over existing low-level optimizers of up to 86% with an average speedup of 5% across 32 benchmarks.

Acknowledgements

None of this would have been possible without the support and encouragement of my advisor, Keith Cooper. He allowed me to thrive and encouraged me to explore the ideas I found interesting. Grad school is much easier when you know your advisor always has your back. I am lucky to have found him. Vivek Sarkar was a great teacher and mentor throughout my time at Rice. I was fortunate to work with him on many different projects and I thoroughly enjoyed our time together. Wotao Yin served as the outside committee member with great enthusiasm. He was a joy to work with and I could not ask for a better participant. I am deeply grateful for everyone that served on my committee.

My friends and fellow graduate students kept me sane during the hard times and happy during the rest. Raghavan Raman, my constant office mate, provided hours of good discussions and taught me everything I know about cricket. Jeff Sandoval and Raj Barik were not only great fellow graduate students, but also great friends. Eddy Westbrook, Ted Dervin, and Rob Banagale provided the friendship that keeps one happy and healthy. Tim Harvey was always there for me. I had many useful conversations with Thomas Schilling about trace-based compilation and Haskell.

I would have never made it without the love and support of Dana Parries, my fiancée. She encouraged me when times got tough and was a constant reminder of the important things in life. I owe her more than I can say, but will still make her call me “Dr. Dave” for the rest of my days.

Finally I would like to thank my family. They have supported and encouraged me my whole life. My parents, Kathleen and David, and my sister Jessica have all positively shaped my future. I have them to thank for all of the advantages I have enjoyed.

Contents

1	Introduction	1
1.1	Organization	2
1.2	Motivation	4
2	Benchmarks	8
2.1	Fibon Benchmark Suite	8
2.2	SPEC Benchmark Suite	12
2.3	Compilers	12
2.4	Machines	15
2.5	Performance	15
3	Low-Level Haskell Code	21
3.1	Example Program	21
3.2	Low-Level Code	26
3.3	Low-Level Control Flow	34
4	Low-Level Haskell Behaviors	39
4.1	Low-Level Behaviors	39
4.2	Measuring Low-Level Behaviors	41
4.3	Results	44
4.3.1	Instruction Mix	44
4.3.2	Branch Mix	47
4.3.3	Indirect Branch Targets	49
4.3.4	Basic Block Length	53
4.4	Conclusion	57
5	Dynamic Trace-Based Optimization of Haskell with DynamoRIO	61
5.1	Hand-Coded Trace Case Study	63
5.2	How DynamoRIO Works	66
5.3	Performance of Applications with DynamoRIO	69
5.4	DynamoRIO Program Traces	75
5.5	Conclusion	84

6	Static Trace-Based Optimization of Haskell with Profile Data	86
6.1	Design	87
6.1.1	Finding Traces	88
6.1.2	Building Traces	100
6.1.3	Optimizing Traces	102
6.2	Implementation	103
6.2.1	GHC Modifications	103
6.2.2	LLVM Modifications	108
6.2.3	Putting It All Together	117
6.3	Results	119
6.3.1	Performance	121
6.3.2	Trace Statistics	129
6.3.3	The Effect of Hotness Thresholds	140
6.4	Conclusion	144
7	Related Work	148
7.1	Static Haskell Optimization	149
7.2	Dynamic Optimization	156
7.2.1	Virtual-Machine Based Optimizers	156
7.2.2	Dynamic Binary Optimizers	162
7.3	Feedback Directed Optimization	166
7.3.1	Collecting Profile Data	166
7.3.2	Optimizing with Profile Data	171
8	Conclusion	174

List of Figures

2.1	Efficiency of the Fibon benchmarks.	11
2.2	Low-level compilation path in the GHC compiler	13
2.3	GHC performance improvement on the Fibon benchmarks.	16
2.4	LLVM performance improvement on the Fibon benchmarks.	18
2.5	Impact of LLVM’s machine independent optimization	20
3.1	Simplified Haskell code listing for sum program	22
3.2	Haskell code listing for sum program	24
3.3	Low-level code for upto entry.	28
3.4	Low-level code for upto continuation point number one	29
3.5	Low-level code for upto continuation point number two	31
3.6	Low-level code for the upto thunk	33
3.7	Hand-coded trace example: non-lazy call graph	34
3.8	Hand-coded trace example: non-lazy call graph	35
3.9	Hand-coded trace example: detailed call graph	37
4.1	Mix of instruction types for Fibon and SPEC programs.	46
4.2	Mix of branch types for Fibon and SPEC programs.	48
4.3	Indirect branch targets for Fibon and SPEC programs.	51
4.4	Indirect branch targets by type for Fibon and SPEC programs.	52
4.5	Indirect branch weighted distribution by type for Fibon and SPEC programs.	54
4.6	Indirect branch target execution percent for Fibon and SPEC programs.	55
4.7	Mix of basic block lengths for Fibon and SPEC programs.	56
4.8	Average basic block lengths for Fibon and SPEC programs.	58
4.9	Average basic block lengths for Fibon and SPECint programs.	59
5.1	Overview of DynamoRIO.	66
5.2	Code Cache Overview of DynamoRIO	67
5.3	Performance of Fibon benchmarks under DynamoRIO	70
5.4	Performance of SPEC benchmarks under DynamoRIO	71
5.5	PC profile results for Fibon benchmarks under DynamoRIO	73
5.6	PC profile results for SPEC benchmarks under DynamoRIO	74
5.7	Average trace length for Fibon benchmarks	77
5.8	Average trace length for SPEC benchmarks	77
5.9	Execution percent of the most frequently executed Fibon traces	78

5.10	Execution percent of the most frequently executed SPEC traces . . .	79
5.11	Number of Fibon traces needed to encompass 50% of execution time .	80
5.12	Number of SPEC traces needed to encompass 50% of execution time .	80
5.13	Sum program with DynamoRIO traces	81
5.14	DynamoRIO traces found for the sum benchmark	83
6.1	The design of the Htrace system.	89
6.2	Algorithm for inserting instrumentation to build program traces. . . .	92
6.3	State transition diagram for trace runtime	94
6.4	Trace runtime callbacks.	96
6.5	The <code>ExtendTrace</code> routine.	97
6.6	The <code>CommitTrace</code> routine.	98
6.7	The shadow tracing routines.	99
6.8	Algorithm for instantiating traces.	100
6.9	GHC <code>build.mk</code> file.	104
6.10	GHC runtime CMM files used in building traces.	105
6.11	GHC <code>rts ghc.mk</code> file.	105
6.12	Tables next to code (TNTC) layout.	107
6.13	Htrace LLVM implementation	109
6.14	External format of trace records.	115
6.15	C type definitions for external trace format.	116
6.16	Preparing a Haskell program to use Htrace	118
6.17	Standard library files used for Htrace programs	120
6.18	Performance of benchmarks under Htrace	122
6.19	Disposition of benchmarks run with Htrace.	123
6.20	Trace code shape for the Dotp benchmark	128
6.21	Number of traces found by Htrace	131
6.22	Types of traces found by Htrace	132
6.23	Types of traces found by Htrace weighted by trace entries	133
6.24	Length of traces found by Htrace	134
6.25	Percent of broken traces found by Htrace	135
6.26	Effect of trace scope on percent of broken traces found by Htrace . .	137
6.27	Weighted average trace completion rate of traces found by Htrace . .	139
6.28	Fibon speedup by hotness threshold.	141
6.29	Distribution of hotness thresholds for best speedup.	143
6.30	Best speedup for any hotness threshold.	144

List of Tables

2.1	Fibon benchmarks	10
2.2	Benchmark machines	15
4.1	Categories used for classifying instruction types.	45
4.2	Categories used for classifying branch types.	47
5.1	Hand-coded trace performance	65
5.2	Key for PC profiling results graphs in Figures 5.5 and 5.6.	72
5.3	Number of traces found by DynamoRIO	82
6.1	List of callback routines for the trace runtime	114
6.2	Benchmark disposition categories	123
6.3	Disposition of benchmark traces.	125
6.4	Sources of improvement for restructured low-level code	128
6.5	Number of traces found by Htrace	130
6.6	Htrace overheads and file sizes	138
6.7	Trace parameters used in the Htrace design.	145

Chapter 1

Introduction

This thesis focuses on the low-level code of Haskell programs. Haskell is a statically typed lazy functional language. While much research effort has gone into improving the execution speed of Haskell, the majority of this effort has focused on high-level transformations that can take advantage of the functional nature of the language. My contributions examine how we can improve performance of Haskell programs by optimizing the low-level code that appears in the translation into assembly code from the high-level Haskell source code. These investigations lead to three major contributions. The first contribution is related to the measurement of low-level Haskell codes, the second contribution examines the possibility of using a binary trace-based optimizer for low-level Haskell, and the third contribution shows how we can improve the performance of Haskell code by using profiles to increase the scope available to a low-level optimizer.

The main contributions of this thesis are in Chapters 4, 5, and 6 that examine the low-level behavior of Haskell programs and explore techniques for improving performance by running low-level optimizations. The remaining chapters provide information about the benchmarks used in the thesis, a look at the structure of low-level Haskell code, and a discussion of related work. The introduction provides an

overview of the organization of the document and the motivation for why this thesis is important.

1.1 Organization

This thesis is organized into five major chapters. We begin by looking at the benchmarks used in experiments throughout the thesis. Next, to clarify what we mean by the phrase “low-level Haskell“, we take a detailed look at an example program. We then take a variety of measurements of the low-level code and compare them to measurements from traditional languages. Finally, we look at trace-based techniques for optimizing the code. One technique uses dynamic binary optimization and the other uses profiling data to statically re-write the code. Each of these chapters is explained in more details below.

Some of the initial investigations in this work used the venerable `nofib` Haskell benchmarks [Partain, 1993]. These benchmarks have served Haskell implementers well for many years, but many of them now run in less than a second which makes it difficult to collect accurate benchmark numbers. This thesis introduces the `Fibon` benchmark suite, which is a new benchmark suite of modern Haskell programs. As part of the work done for this thesis, we created the `Fibon` benchmark suite for evaluating the effects of compiler optimizations. Chapter 2 discusses the composition of the `Fibon` benchmark suite and contains some performance measurements on the effectiveness of low-level optimizations on Haskell programs. Our experiments show that low-level optimizations are generally ineffective for Haskell codes. To understand the root of the difficulties in applying low-level optimizations to Haskell, we try and compare the low-level code of Haskell and other languages.

To get a better sense of how the low-level code of Haskell programs compares to the low-level code of traditional languages like C, C++, and Fortran, we measure the

behavior of programs from the Fibon and SPEC benchmark suites. Chapter 4 discusses how we measure the low-level behavior of programs and shows the comparison between Haskell and traditional languages. The study reveals some differences in the behaviors that we measured for these low-level codes, particularly with the amount of indirect control-flow operations. We can better understand these differences in the context of a detailed example of low-level Haskell code.

Chapter 3 provides a detailed look at a Haskell program and its translation to low-level code. This examination shows how the abstraction mechanisms of Haskell make it difficult to perform low-level optimizations. A key problem with trying to optimize Haskell programs is that the program control flow is obscured by the use of high-order functions and lazy evaluation. Although the control flow is difficult for the compiler to deduce statically, it is readily available at runtime. We looked at two different techniques for finding program traces and using them to give the compiler a larger scope for optimization: dynamic binary and profile-guided tracing.

DynamoRIO is a binary trace-based optimization system developed by Bruening et al. [2003]. It takes an unmodified program and builds traces of frequently executed paths with the option to run additional optimizations when a trace is built. The improved scope of a program trace would appear to be a good fit for low-level optimization of Haskell programs. Chapter 5 examines how Haskell programs perform when running under DynamoRIO. While DynamoRIO is able to find traces in Haskell programs, the overhead of running under DynamoRIO outweighs any potential benefits. The fact that DynamoRIO is able to find program traces is encouraging. Our experience with DynamoRIO led us to explore techniques that would move the trace-building and optimization offline. That led to our work on statically derived, profile-based traces and their optimization.

Chapter 6 presents Htrace, which is a system we built for profile-guided optimization of Haskell programs. Since the profiling and optimization are done separately

from the normal program execution, we avoid the overheads associated with building traces at runtime. Htrace runs the program once to find hot traces in the program and then uses these traces as an increased scope for optimization. The end result is a system that increases the performance of Haskell programs by up to 86% for programs that contain frequently executed loop-based traces. For other programs, the traces have little effect or can sometimes harm performance. This chapter shows that the performance of Haskell programs can be improved by focusing solely on the low-level code without knowledge of the high-level structure of the Haskell program.

A reader of this thesis is not expected to have an in depth knowledge of Haskell. Familiarity with a functional programming language would be useful, but most of the technical discussion focuses on the low-level code produced during the compilation process. This low-level code is certainly shaped by the fact that it comes from Haskell, but it can be understood as a unique artifact without having to understand the execution model of Haskell. Chapter 3 contains a detailed example of high-level functional Haskell code and its translation to the low-level imperative code that is the focus of this thesis. The information in that chapter should be enough background to understand the important features of Haskell as they relate to this thesis.

1.2 Motivation

Software is important and ubiquitous. We rely on software to improve our lives and keep us safe. Software is used in many vital human activities such as transportation, commerce, and scientific discovery. Yet even though software is already widespread, it is likely to continue to gain importance as we rely on it to further automate our lives and expand the boundaries of knowledge. As software continues to push itself into our lives the importance of writing correct and maintainable code grows. Consider an example from the scientific domain.

A survey by [Hannay et al. \[2009\]](#) found that scientists spend 30% of their time developing software, and the amount of time spent developing software has increased compared to ten years ago. Computation is now an integral component in scientific research so we must find a way to leverage the increasing power of computer hardware without burdening scientists with onerous programming tasks. [Hannay et al.](#) additionally report that most scientists develop software on desktop computers with fewer than 10% of scientists targeting a supercomputer. These results suggest that productive programming languages are important and that software developed for desktop computing is an important subject of focus. Despite their increased importance in scientific discovery, programs are still difficult to write correctly. A recent article in Nature [[Merali, 2010](#)] describes several situations where software bugs have led to retracted publications.

These examples point to the importance of software and the difficulty of writing correct programs. While the cited examples come from scientific domain, the difficulty of writing correct software is certainly not isolated to scientific computing. If we are to meet the broad set of challenges of the future across all knowledge domains we must provide programmers with languages that are expressive, safe, scalable, and efficient.

Haskell is a lazy functional programming language that is well suited to writing concise and correct programs. It is expressive because of its declarative nature, which allows the programmer to specify *what* is to be computed rather than the exact steps for *how* the computation should be carried out. It is safe because the strong static type system prevents many programming errors at compile time and allows components to be safely reused. Haskell is a purely functional language, which means side effects are easily controlled and the programmer can reason about the correctness of disjoint parts of a program. It is scalable because of the many abstractions it provides for building large programs: higher-order functions, lazy evaluation, and type classes.

Higher-order functions allow computations to be explicitly represented as data and provide a standard way to abstract over computations. Lazy evaluation lets the programmer write program definitions without worrying about the exact order in which they will be executed and allows the programmer to cleanly separate the process of producing and consuming data. Finally, type classes are used to express ad-hoc polymorphism and provide similar abstraction mechanisms as interfaces or abstract classes in object-oriented languages. Unfortunately, all these abstractions can harm performance which may dissuade programmers from using the language.

It is the job of the compiler to translate high-level abstractions to low-level machine code with a minimal loss in performance. The programmer may be willing to tolerate a decrease in performance for an increase in productivity. However, there is a limit to how much performance one is willing to pay for these high-level abstractions. When the performance cost is too great, a programmer will switch to a different language with fewer abstractions but better performance. Ideally we could have a language with all of the high-level abstractions and great performance. In this thesis I seek to advance the state of the art in optimizing functional languages by employing dynamic low-level compiler optimizations. My goal is to further close the performance gap between high-level functional languages and low-level imperative languages.

Although performance is an important consideration, there are many factors that influence the popularity of a programming language. Languages that are commonly taught and used in university courses are generally more successful because of the large number of programmers that know how to use the language. Education, quality libraries, and corporate support are all important factors in language adoption. Although functional programming languages, such as Haskell, may have many good qualities they will only thrive when these good qualities are widely recognized. Fortunately, multi-core computing has brought a renewed interest to functional programming languages because of their suitability for parallel programming.

I believe that the safe and expressive nature of functional programming languages will be important for the future hardware trend of mainstream parallel computers. My work will encourage the adoption of functional programs in domains where performance remains an important metric for success. My work seeks to improve the performance in a single thread of control and any parallelism in the program would automatically take advantage of the improvements found in my research. Even if Haskell does not see a widespread adoption in the scientific community, the focus of my work on reducing the cost of abstractions through low-level optimization will provide a valuable reference point for future implementations of expressive and efficient programming languages.

Chapter 2

Benchmarks

In this chapter we describe the Fibon benchmark suite, a new Haskell benchmark suite developed during the production of this thesis. The primary motivation for developing a new benchmark suite came from the quick running time of many programs from the popular `nofib` benchmark originally introduced by [Partain \[1993\]](#). We wanted a benchmark suite that contained programs that ran for more than one second so that the impact of low-level optimizations could be reliably measured. In this chapter we will describe the Fibon benchmark suite and provide basic performance measurements for the effectiveness of low-level optimizations. Although the Fibon benchmark suite was developed for this thesis, none of the benchmarks were written by the thesis author. Each of the individual benchmark programs predates this thesis.

2.1 Fibon Benchmark Suite

The Fibon benchmark suite is a collection of 32 benchmarks from four different sources [Fibon \[2010\]](#). Each source provides a different outlook on the use of Haskell. The four benchmark *Groups* are described below and the benchmarks from each group are summarized in [Table 2.1](#).

Hackage [Hackage \[2010\]](#) is an open source repository for all kinds of Haskell codes.

The programs come from a wide variety of domains, including: computer algebra, compression, cryptography, graph algorithms, linguistics, SAT solving, finite automata, parsing, scientific simulation, and bioinformatics.

Shootout The [Shootout \[2009\]](#) benchmarks were created to compare performance across a wide variety of programming languages. The benchmarks are small programs, but they have been written specifically to run fast.

Repa Repa is a library for parallel arrays in Haskell written by [Keller et al. \[2010\]](#). The benchmarks are small programs that test the quality of the code generated for the library. These programs tend to be loop-based codes that work on the parallel arrays.

DPH Data Parallel Haskell (DPH) is a Haskell library for nested data parallelism by [Chakravarty et al. \[2007\]](#). The benchmarks are small programs that test the quality of the code generated by the library. These programs are similar to the Repa group in that they tend to be loop-based codes working on arrays.

In addition to the informal characterization of the benchmark category given in [Table 2.1](#), an important characteristic to measure for Haskell benchmarks is the amount of time spent doing garbage collection compared to the amount of time spent executing the program. In a Haskell implementation, the *mutator* is the code that does the real work of the program (e.g. computing π to 1000 digits) and the garbage collector is an implementation detail that is only needed because our computers have finite memories. The *efficiency* of a Haskell program is the ratio of the time spent running in the mutator compared to the total execution time. A low efficiency means that a lot of execution time was spent collecting garbage.

[Figure 2.1](#) shows the efficiency of the Fibon benchmarks as executed by GHC. The Hackage group shows the greatest variety in efficiency, from the very inefficient

Benchmark	Category	Lines of Code
Hackage		
Agum	Algebra	786
Bzlib	FFI	432
Cpsa	Cryptography	11582
Crypto	Cryptography	4486
Fgl	Graphs	3834
Fst	Compilers	4532
Funsat	SAT	16085
Gf	Linguistics	23972
HaLeX	RE	4035
Happy	Compilers	5837
Hgalib	AI	822
Palindromes	String	496
Pappy	Compilers	7313
Regex	RE	6873
Simgi	Simulation	5134
TernaryTrees	String	722
Xsact	Bioinformatics	2783
TOTAL		104221

Benchmark	Category	Lines of Code
Dph		
_DphLib		316
Dotp	Array	308
Qsort	Array	236
QuickHull	Array	680
TOTAL		1612
Repa		
_RepaLib		8775
Blur	Array	106
FFT2d	Array	89
FFT3d	Array	103
Laplace	Array	274
MMult	Array	133
TOTAL		9480
Shootout		
BinaryTrees	GC	92
Chameneos	Threads	96
Fannkuch	Array	27
Mandelbrot	Array	68
Nbody	Array	192
Pidigits	Stream	26
SpectralNorm	Array	97
TOTAL		598

Table 2.1: The list of Fibon benchmarks. The Hackage benchmarks are the most diverse since they come from a wide variety of sources. The Dph and Repa benchmarks are pulled from other benchmark suites that focus on array-oriented codes. The Shootout benchmarks are small programs mostly have array-oriented programs, but also contain several non-array codes.

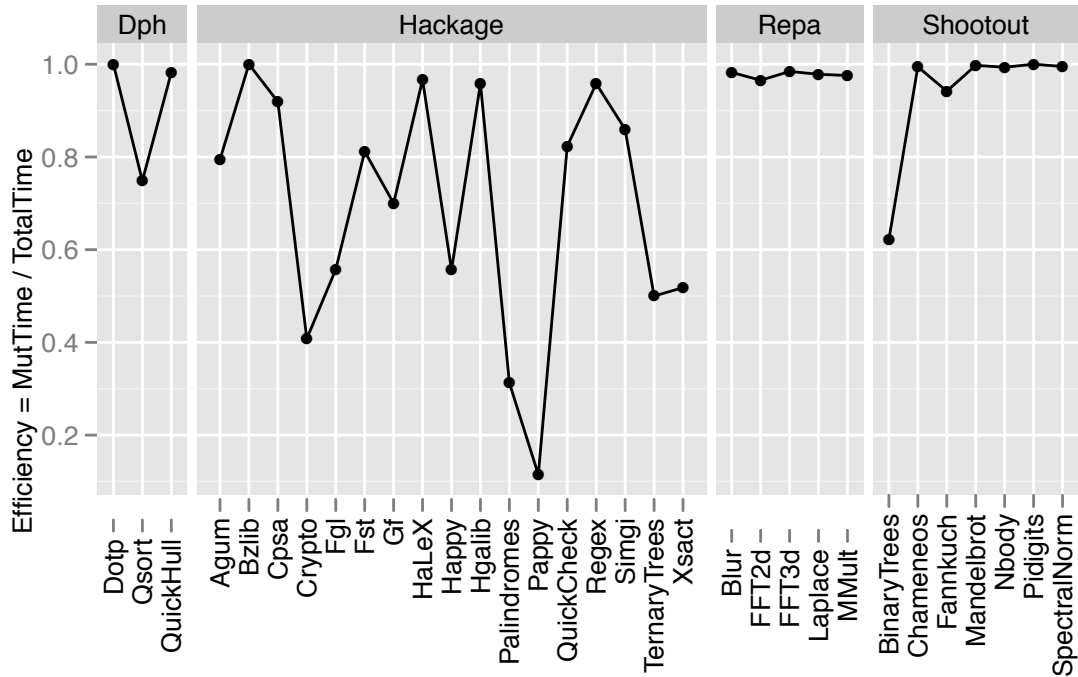


Figure 2.1: Efficiency of the Fibon benchmarks. The efficiency is a measure of the time spent in the program mutator compared to the total execution time. A low efficiency means that a lot of execution time was spent in the garbage collector.

Pappy benchmark at around 10% to the very efficient Bzlib benchmark at near 100%. The Repa benchmarks all have a very consistently high efficiency. The Shootout benchmarks have high efficiency with the exception of the BinaryTrees benchmark, which was actually designed to test the performance of the garbage collector.

In this thesis we focus on low-level optimization opportunities. These optimizations seek to improve the execution time of the mutator rather than change the behavior of the garbage collector. To reflect that focus, most of the runtime numbers we present in this thesis measure changes in the mutator execution time. The overall impact of these optimizations will depend on the efficiency of the program, but measuring changes in mutator time allow us to detect performance improvements even in benchmarks that have low efficiency. Although there are many high-level optimizations that can affect GC performance (e.g. by allocating fewer bytes), the low-level

optimizations we target in this thesis focus on mutator performance.

2.2 SPEC Benchmark Suite

The SPEC benchmark suite is an industry standard for evaluating the performance of a computer system [SPEC, 2006, SPEC CPU Subcommittee, 2006]. The benchmark suite is self-described as a “next-generation, industry-standardized, CPU-intensive benchmark suite, stressing a system’s processor, memory subsystem and compiler.” The CPU benchmarks are compute-intensive benchmarks that were developed from real applications, and are divided into two groups *SPECint*, and *SPECfp*.

The SPECint group contains benchmarks testing integer performance and the SPECfp group is for testing floating point performance. The benchmarks are widely used in evaluation of compiler optimizations. In this thesis, we use them as a reference set of programs from traditional languages and as a foil for the Fibon Haskell benchmarks. The SPECint benchmarks are all written in C and C++ while the SPECfp benchmarks contain a mix of C, C++, and Fortran.

2.3 Compilers

This thesis revolves around two different compilers: GHC and LLVM. In this section we discuss the basic details of the two compilers.

The Glasgow Haskell Compiler (GHC) Peyton Jones and Marlow [2011] is a state-of-the-art compiler for Haskell. It contains a large number of high-level optimizations for improving the execution time and reducing the space usage of Haskell programs. In addition to the standard Haskell language, GHC includes a number of popular extensions used when writing modern Haskell programs. More information on the high-level optimizations used by GHC is given in Chapter 7.

The results in thesis are based on GHC version 7.4. The compiler has largely

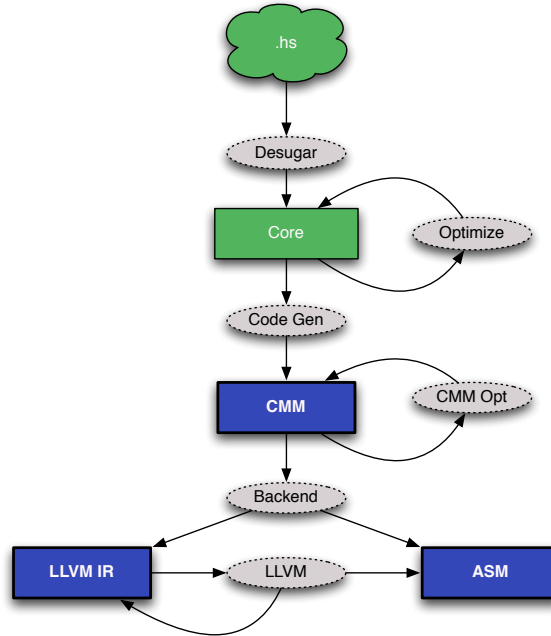


Figure 2.2: Low-level compilation path in the GHC compiler. The square boxes are different program intermediate representations. The oval shapes represent different transformations that work on the various intermediate representations.

been used without modification. The primary exception is in Chapter 6 where the compiler was slightly modified to enable easier collection of program traces. One major advantage of using GHC is its inclusion of an LLVM back end, which was added by Terei and Chakravarty [2010]. The LLVM back end is an alternate to the native code generator which directly outputs assembly code. The compilation path in GHC is shown in Figure 2.2.

The actual compilation path in GHC is obviously much more complicated than the simplified picture shown in the figure. However, the simplified view is sufficient for understanding the work in this thesis. The high-level optimizations are left untouched by this work so that we can take advantage of the many optimizations used by GHC. The high-level optimizations are represented by the Core-to-Core transformation in the image. The Core language is then lowered to CMM through a transformation called CodeGen in GHC. CMM is low-level intermediate representation. GHC con-

tains a few simple local optimizations for CMM such as constant propagation. From the CMM, GHC will go through one of two back ends: the native back end or the LLVM back end. The native back end will directly generate assembly code from the CMM. The LLVM back end will generate LLVM IR from the CMM and then use LLVM to optimize the IR and generate native code.

LLVM is a compiler infrastructure dedicated to “lifelong program analysis and transformation” [Lattner and Adve, 2004]. The results in this thesis are produced with LLVM version 3.0. The LLVM compiler was modified to build program traces as described in Chapter 6. LLVM has gained popularity in recent years because of its large number of program transformations and (perhaps more importantly) its permissive open source license. The LLVM framework contains several tools that combine to optimize the LLVM IR and generate machine code. The main tools used in this thesis are `opt`, `llc`, and `lli`.

The `opt` tool is LLVM’s machine independent optimizer. The current version of LLVM lists over 100 different analysis and transformation passes available [LLVM, 2011]. These optimizations include many of the classic compiler optimizations found in compiler text books like Cooper and Torczon [2012].

The `llc` tool is LLVM’s machine dependent optimizer. These optimizations include standard compiler back end optimizations such as instruction selection, register allocation, and instruction scheduling. Additionally the `llc` tool has optimizations that become available as the LLVM IR is lowered to machine code, such as strength reduction of array address calculations.

The final important tool for this thesis is the `lli` tool which is used to interpret and JIT compile LLVM IR files. The `lli` tool is important for building program traces and its use is more fully described in Chapter 6.

Chapters	4, 5	3, 6
Machine		
Model	Dell T5400	Macbook Pro
Year	2009	2010
RAM	8GB	8GB
OS		
Flavor	Linux	Mac OS
Version	Fedora 14	10.7.2
Kernel	2.6.35.6-45	11.2.0
Arch	x86_64	x86_64
CPU		
Model	Intel Xeon E5405	Intel Core i7
Speed	2.00GHz	2.66GHz
Processors	2	1
Cores	$8 = 4 \times 2$	2

Table 2.2: Benchmark machines

2.4 Machines

Two different machines were used to collect results for this thesis. The machines are listed in Table 2.2. The T5400 was used to collect the low-level behavior measurements in Chapter 4 and for the DynamoRIO results in 5. The Macbook was used to generate the example code for Chapter 3 and for the LLVM-based optimizer described in Chapter 6.

2.5 Performance

The performance characteristics of the Fibon benchmarks are discussed in this section. We look at two main characterizations of performance. First, we look at the effectiveness of high-level optimizations done by GHC. We will see that GHC does a very good job at optimizing Haskell programs and can achieve a speedup of up to 97× over a non-optimized program. Second, we look at the effectiveness of the low-level optimizations implemented by LLVM. We will see that generally the LLVM optimizations have little effect on the performance of Haskell programs, with most

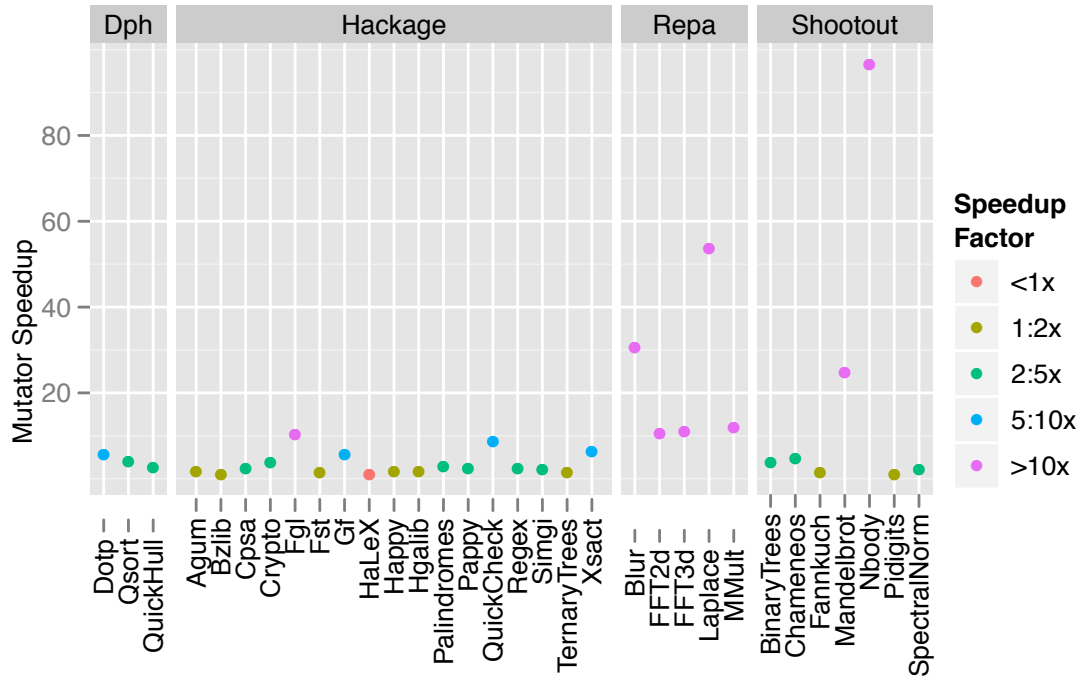


Figure 2.3: GHC performance improvement on the Fibon benchmarks. This graph shows the speedup GHC achieves comparing optimization level `-00` vs. `-02`. The high-level optimizations in GHC are generally very effective, with a whopping $97\times$ speedup on the Nbody benchmark. The average (geometric mean) speedup is $4.28\times$.

benchmarks exhibiting a performance change of less than 5%.

The high level optimizations done by GHC are generally very effective for the Fibon benchmarks. Figure 2.3 shows the speedup GHC obtains using its optimizations. It measures the benchmark performance when running GHC with no optimizations (`-00`) compared to running with a high optimization level of `-02`. We can see that the high-level optimizations performed by GHC are generally very effective. The HaLeX benchmark is the lone outlier in that it runs about 5% slower. Many benchmarks achieve a speedup of $2\times$ or more. The Repa benchmarks in particular see a large speedup with each benchmark running at least $10\times$ faster. Although the speedups from the high level GHC optimizations are quite impressive, the low-level optimizations done by the LLVM compiler tell a different story.

Figure 2.4 shows the performance benefit we get from running the Haskell programs through the LLVM compiler. The graph compares the performance of the program when compiled using GHC’s native code generator to the performance when compiled with the LLVM code generator. The same GHC high-level optimizations were used in both cases. As we can see the majority of the programs show little improvement from running through the LLVM back end. The dearth of improvement is somewhat surprising given the large number of optimizations performed by LLVM compared to the GHC back end. Four of the programs show an improvement of greater than 20%: Blur, Laplace, Mandelbrot, and Nbody.

The four benchmarks that show the greatest improvement all have a very special property in common that none of the other benchmarks share. Each of these benchmarks have loops that LLVM can identify. These loops are single tail-recursive functions. Although the Fibon benchmarks have many programs that have tail-recursive loops in the Haskell code, these loops are obscured by the translation process which inhibits the optimization efforts of LLVM. A major goal of this thesis is to make the loops in the program visible to LLVM so that it can optimize them to the same degree as these four benchmarks. In the end, we were able to expose low-level Haskell loops for many of the Fibon benchmarks, and the performance improvements we report in this thesis come from exposing these loops to LLVM, which is able to optimize many, but not all, of the loops we exposed.

We can also examine how the optimization level used by LLVM effects the performance of our Haskell benchmarks. Figure 2.5 shows the speedup we achieve on the Fibon benchmarks using only the machine-independent optimizations of LLVM. The programs were first compiled with `opt -O0` and compared against `opt -O2`. The same set of back end optimizations were used by the `llc` program.

We could not separately test the effects of different levels of back end optimization since the programs all crashed when running with `llc -O0` and the levels `-O1` and

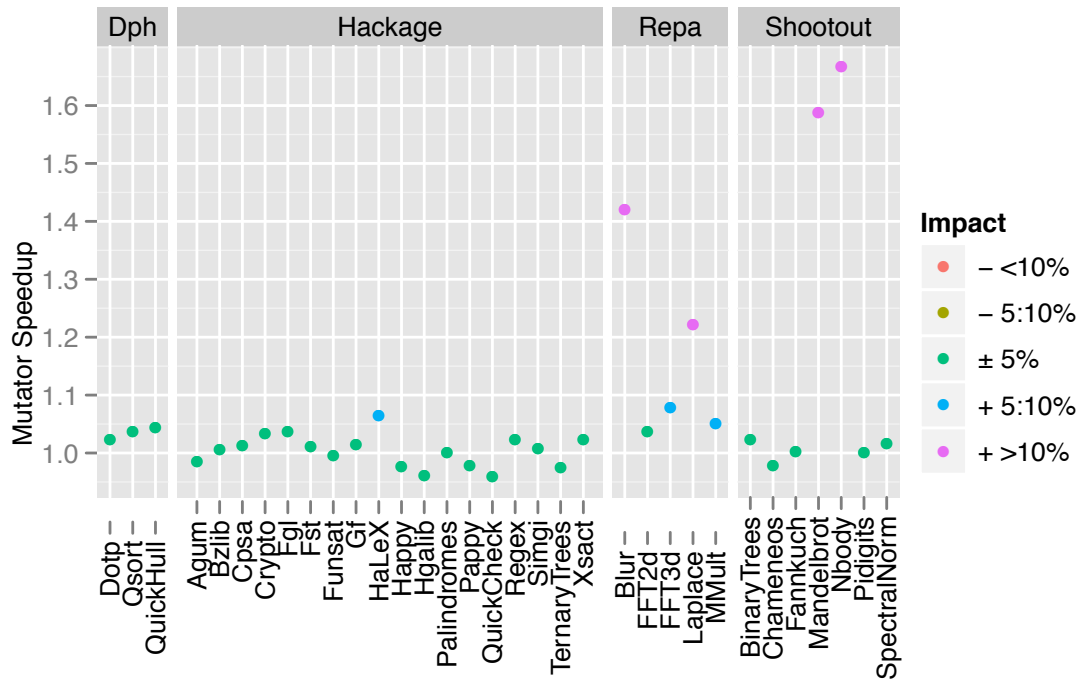


Figure 2.4: Speedup of Fibon benchmarks when optimizing with LLVM. This graph shows the speedup we get by running the Fibon benchmarks through the LLVM compiler compared to using GHC’s native back end. The majority of the programs get little benefit by running through LLVM. The four programs that get a large speedup all have a similar property: they have loops that are contained in a single function. The average (geometric mean) speedup is 5.8%.

-02 run the same sequence of optimizations. The crashes were likely due to the need to run the register allocator for these programs since GHC uses a custom calling convention in LLVM which passes many parameters in registers. Still, we can gain some insights by examining the effects of the machine independent optimizations.

The most pressing observation is that the four loop-based benchmarks that had the largest speedups compared to GHC’s back end (Blur, Laplace, Mandelbrot, Nbody) do not show a great improvement from the `opt`-based optimizations. The Blur, Laplace, and Mandelbrot benchmarks achieve a speedup of less than 5% from the `opt`-based optimizations. The Nbody benchmark improves slightly less than 10%. These numbers indicate that the majority of the improvement that LLVM gains is from the machine-dependent optimizations present in the `llc` tool.

Since GHC’s native back end performs no loop optimizations, we can identify the optimizations that are the likely cause of improvement in the loop-based benchmarks. GHC uses a simple register allocator that is unlikely to capture any induction variables in the loops causing the registers to be shuffled at the end of each loop iteration. Additionally, the `llc` tool contains a code motion algorithm [Knoop et al., 1994] to move operations out of the loop and a strength reduction algorithm [Cooper et al., 2001] to reduce the cost of updating loop induction variables. These optimizations are the major scalar optimizations we can perform on loops and since they are all present in the `llc` tool the machine-independent optimizations in `opt` do not add any great benefits to the loop-based benchmarks.

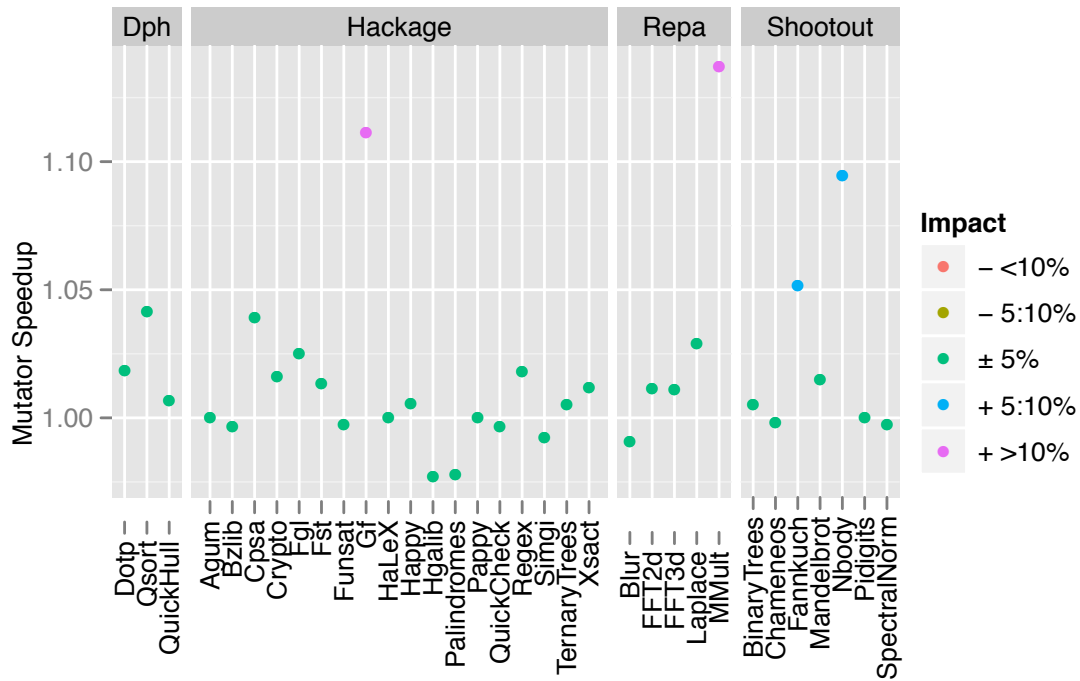


Figure 2.5: Impact of LLVM’s machine independent optimizations (`opt -O2` vs `opt -O0`). This graph compares the effect of different levels of machine independent optimizations in LLVM (the optimizations run by the `opt` program). The machine-independent optimizations do not have as great an impact as the back end optimizations as seen by the small speedup in the loop-based benchmarks. The average (geometric mean) speedup is 1.7%.

Chapter 3

Low-Level Haskell Code

In this chapter we take a detailed look at the low-level code that gets generated for Haskell programs. We focus on a simple example so that we can fully see the structure of the low-level code. We start by describing a Haskell source program for readers unfamiliar with Haskell. Next we describe the low-level code that is generated for a single function in the program. Finally, we explore the control flow in the program and identify a simple optimization that could be done by the compiler if it had more information about the runtime paths executed by the program.

3.1 Example Program

The program used in the case study is shown in Figure 3.1; it computes the sum of integers from 1 to 300,000,000. The code is written in a functional style and uses the type-class and lazy-evaluation abstraction mechanisms. There are two primary functions involved in the computation: `upto` and `sum`. The `upto` function lazily generates a list of the numbers. This list is consumed by the `sum` function, which is a tail-recursive function that uses an accumulator parameter to hold the result of the sum.

The syntax of Haskell may be a bit foreign to programmers coming from traditional

```

1 root :: Int -> Int
2 root x = sum (upto 1 x)
3
4 upto :: Int -> Int -> [Int]
5 upto from to =
6     if from > to then [] else from : upto (from + 1) to
7
8 sum :: (Num a) => [a] -> a
9 sum l = sum' l 0
10     where sum' []      a = a
11           sum' (x:xs) a = sum' xs (a+x)

```

Figure 3.1: Simplified version of the example program used in this chapter. See the text for a description of the syntax. The full version of the program is shown in Figure 3.2.

languages, so we will give a detailed explanation of the code in this figure. The overall computation is captured by the `root` function. Line 1 contains the type signature of the function. The double colon `::` is read “has type”, and is used to give a type signature. The arrow `->` represents a function type, so the type `Int -> Int` is a function that takes one integer parameter and returns an integer. Line two is the definition of the `root` function. The parameter to the function is the variable `x`, and it computes the `sum` of values from one `upto x`.

The `upto` function is defined on lines 4-6. It computes the list of integers in the range `from` up to `to`. Line 4 contains the type signature, which says that `upto` is a function that takes two integers and returns a list of integers. The square brackets `[]` are used to represent the type of singly linked lists in Haskell. The definition of the function is given in line 6, and it shows how the resulting list of integers is computed. If the `from` parameter is greater than the `to` parameter then the empty list is returned, denoted by the empty square brackets. Otherwise, a new list element is returned. The colon operator `:` read as “cons” constructs a new list element. At the head of the list is the current value of `from`. The tail of the list is computed as a recursive call to the `upto` function with the current `from` incremented by one.

Because Haskell is a lazy language, the recursive call is not performed immediately, but rather it is delayed until the tail of the list is demanded.

The `sum` function is the final piece of the program. Line 8 is the type signature of the function, which indicates it is a polymorphic function. The `sum` function works for any type `a`, as long as `a` belongs to the `Num` type class. The `Num` type class requires that the type implement standard numerical operations (such as addition). The `sum` function takes a list of the `Num`-implementing values and returns a single value which is the sum of the list. The function uses an auxiliary function defined in lines 10-11 to actually compute the sum. The auxiliary function is defined by case analysis on the list. If it gets an empty list, then the accumulated sum is returned. Otherwise, the head of the list is added to the accumulated sum and we recursively compute the sum for the tail of the list.

The code in Figure 3.1 shows the basic program we use in this chapter. In order to get presentable low-level code we can use to explain Haskell behavior, we actually use a more detailed version of the code shown in Figure 3.2.

There are several interesting features of the detailed code in Figure 3.2. The `upto` function is not polymorphic because it can only generate a list of `Int` values which is Haskell's representation of boxed machine-sized integers. A Haskell `Int` is a boxed wrapper around an unboxed integer. GHC uses a naming convention where types and operators that work with unboxed types end in a `#`. The `upto` function is written using GHC-specific primitive functions for operating on the unboxed part of the integers (i.e. the `m` and `n` values inside the `I#` boxes). Using the primitive operations (`>#`, `+#`) simplifies the generated code and makes the example easier to follow.

The `sum` function is polymorphic because it can sum a list of any type belonging to the `Num` type class (e.g. `Int`, `Float`, `Double`). The actual implementation of the `Num` class specifies how to perform the addition in the `sum` function (`a+x`) and


```

1 {-# LANGUAGE NoImplicitPrelude, BangPatterns, MagicHash #-}
2 module Main where
3
4 import GHC.Base
5 import GHC.Num
6 import qualified Prelude
7
8 {-# NOINLINE upto #-}
9 upto :: Int -> Int -> [Int]
10 upto from@(I# m) to@(I# n) =
11     if m ># n then [] else
12     from : upto (I# (m +# 1#)) to
13
14 {-# NOINLINE sum #-}
15 sum :: (Num a) => [a] -> a
16 sum    l          = sum' l 0
17     where
18         sum' []      !a = a
19         sum' (x:xs) !a = sum' xs (a+x)
20
21 {-# NOINLINE root #-}
22 root :: Int -> Int
23 root x = sum (upto 1 x)
24
25 main = do
26     let res = root (I# 300000000#)
27     Prelude.putStrLn (Prelude.show res)

```

Figure 3.2: Complete code listing that produces the traces in Figure 3.9. The program computes the sum of the first 300,000,000 integers. The integers are generated lazily by the `upto` function, and the sum is computed using a strictly evaluated accumulator as a tail recursive function.

how to convert the constants (e.g. 0) to a value of the appropriate type. The `Num` implementation is passed to the `sum` function as a dictionary that holds pointers to the `Num` functions for that type. When a method of the `Num` type class is called (e.g. `+`), the generated code will lookup the appropriate function in the dictionary. This function is then called using the generic apply routine in the GHC runtime [Marlow and Peyton Jones, 2006].

I should also note that `sum` uses the auxiliary function `sum'` to consume the lazily generated list. This function uses an accumulator parameter that stores the sum as the list is consumed and recursively calls itself with the updated parameter. The `sum'` function is strict in the accumulator parameter (seen by the `!` in front of the `a`) so that the addition will be performed before the recursive function call. If `sum'` was not strict in `a`, then the addition would be delayed as a thunk and not performed until the entire list had been consumed.

The final point to note about the detailed example program is that it has been purposely annotated with `NOINLINE` pragmas to keep GHC from inlining the functions and optimizing away the example. The example program was selected because it is simple, which makes it easy to manipulate by hand. GHC can optimize away most of the overhead in this example by inlining the functions, but the purpose of this exercise is to show the problems with code generated by GHC and to show how trace-based optimization can improve the performance of the code. Although GHC can easily optimize this program, we use it to show the potential of trace-based optimization because the program contains overheads common to Haskell programs that GHC may not be able to so easily optimize when they are obscured in more complex or convoluted code.

3.2 Low-Level Code

In this section we take a look at some of the low-level code that is generated for the example program. We will focus on the code generated for the `upto` function to keep the discussion manageable. The discussion here focuses on the parts of the Haskell code that is most relevant to this thesis. We do not attempt to give a full description of the execution model used by GHC to implement Haskell programs. GHC uses the Spineless Tagless G-machine (STG) execution model described by [Peyton Jones \[1992\]](#). The description in that paper is fairly accurate for today's GHC. Some major changes include the use of `eval/apply` for calling unknown functions [[Marlow and Peyton Jones, 2006](#)] and the addition of pointer tagging to avoid unnecessary indirect jumps [[Marlow et al., 2007](#)].

One important feature of GHC is that it manages the Haskell call stack separately from the standard C stack. During the translation from the high-level Haskell code to the low-level imperative code, the stack manipulations are made explicit in the low-level code. One implication of managing the Haskell call stack separately is that when a call is made to another function, GHC must generate a return point where the call can return. These return points serve to hold the code for the continuation as well as an indication of which values are pointers of the values that have been saved on the stack for use in the continuation. The pointer info is needed by the garbage collector to accurately find all of the roots for garbage collection. The return points cause a single source function to get broken up into multiple pieces. For example, the `upto` function is broken into four pieces.

To understand the low-level code we need to know a bit about the calling convention GHC uses for Haskell programs. The calling convention is used for the code at the CMM level and lower (see [Figure 2.2](#)). GHC compiles programs with a number of virtual registers `R1-RN`. These registers are assigned specific purposes on entry to a function. The `R1` register contains the value under evaluation. For a function this

is simple a pointer to the function, but in some instances it will point to a *thunk*. A thunk is simply a value whose evaluation has been delayed to respect Haskell’s lazy evaluation strategy. The registers `R2`, `R3`, `...` contain the arguments to the function. In addition there are several special registers: `Sp`, `Hp`, `SpLim`, `HpLim`, and `BaseReg`. The `Sp` register contains a pointer to the top of Haskell runtime stack, and `SpLim` contains the stack limit. Similarly, `Hp` contains a pointer to the next free memory location in the heap and `HpLim` is the current heap limit. These heap registers allow very a cheap allocation strategy of simply bumping the current heap pointer to allocate memory. The notation `I64[x]` represents a dereference of a memory location that is 64 bits wide and is pointed to by the value `x`. It is used as the notation for both loading and storing values to memory. Finally, the `BaseReg` register points to a table of commonly used function addresses, such as the garbage collector entry point.

The `upto` function must perform several tasks: ensure that the arguments are evaluated, check to see if the limit has been reached (`from > to`) and if not to allocate a new list element in the heap. Figure 3.3 shows the CMM code for the entry to the `upto` function. Because Haskell uses lazy evaluation the arguments to the `upto` function may not be evaluated yet. The entry code ensures that the first argument is evaluated and hands control to the continuation that will finish the remaining tasks. On entry to the function we first check to make sure that we have enough space on the runtime stack (line 4) and if not we call back into the runtime to extend our stack (line 13). Next we prepare to evaluate the first argument, which is the `from` variable stored in `R2`. Because of lazy evaluation, evaluating the argument may require an unknown amount of computation. We need to save any live variables to the stack so that we can access them when we need them later. Line 5 saves the second function argument (`to` stored in `R3`) to the stack. Line 7 pushes the continuation point (`sMX_ret`) onto the stack so that we will return to the correct place after evaluating the first argument. Now we are ready to evaluate the argument.

```

1 upto_entry()
2 {
3     cQ2:
4         if ((Sp + -24) < SpLim) goto cQ4;
5         I64[Sp - 8] = R3;
6         R1 = R2;
7         I64[Sp - 16] = sMX_ret;
8         Sp = Sp - 16;
9         if (R1 & 7 != 0) goto cQ7;
10        jump I64[R1] ();
11    cQ4:
12        R1 = Main.enumFromTo_closure;
13        jump (I64[BaseReg - 8]) ();
14    cQ7: jump sMX_ret ();
15 }

```

Figure 3.3: Low-level code for upto entry. It evaluates the `from` argument.

In line 6 we copy the first argument into `R1` since that will be item under evaluation. We can now evaluate the argument. GHC represents values uniformly as a closure that contains a pointer to the code that will evaluate the value along with any free variables needed to evaluate the value. To evaluate a value we simply jump to the code pointer stored in the closure as shown in line 10. As an optimization, GHC will tag closures that have already been evaluated by marking the low-order bits in the address pointing to the closure [Marlow et al., 2007]. Before jumping to the closure, GHC will first check to see if it has been tagged as evaluated as shown in line 9. If the closure is already evaluated we can jump directly to the continuation point as shown in line 14. Either way we will eventually end up at the `sMX_ref` continuation with the first argument evaluated.

Figure 3.4 is the continuation point for the `upto` function after evaluating the first argument. It has a similar structure to the entry point described above and is responsible for evaluating the second argument. First, on line 4, we save the value that was under evaluation to the stack. This value is actually the first argument to the `upto` function that was evaluated by the entry code. It is saved on the stack

```

1  sMX_ret()
2  {
3      cPS:
4          I64[Sp + 0] = R1;
5          _cPQ::

```

Figure 3.4: Low-level code for `upto` continuation point number one. It evaluates the `to` argument.

because we must now evaluate the second argument to the `upto` function. In line 6 we save the primitive integer value of the `from` variable to the stack. Recall from Section 3.1 that in Haskell integers are stored as boxed values in the heap. Here, we extract the unboxed integer and save it on the stack for later use. Line 5 loads the `to` variable that we saved on the stack in the entry code. The variable is loaded into the `R1` register in preparation to evaluate it. Line 8 saves the continuation address (`sNO_ret`) to the stack so that we return to the correct location after evaluating the value in `R1`. Finally, in line 10 we check to see if the `to` value has already been evaluated. If it is already evaluated we jump to the continuation, otherwise we jump to the code that evaluates the parameter.

After evaluating both of the arguments to the `upto` function we end up at the `sNO_ret` continuation shown in Figure 3.5. This continuation corresponds to the part of the `upto` function that checks the termination condition and then returns the next list element or the empty list if the limit has been reached. If the limit has not been reached then the head of the list returned to the caller contains the current number in the enumeration (the `from` argument to the `upto` function) and the tail of the

list contains a thunk that can be evaluated to produce the rest of the list. This two element combination is the prototypical lazily evaluated list.

On entry to the continuation the `R1` parameter contains the `to` argument of the `upto` function. In lines 4-5 we check to make sure there is enough space in the heap to allocate the list element and the thunk. If not, we call the garbage collector in line 19. Line 6 is the check to see if `m > n`. The comparison uses the unboxed integer values stored on the stack for `from` and in the `to` closure pointed to by `R1`. If the limit is reached the empty list is returned to the caller in lines 21-24. Otherwise, we allocate a new list element and thunk.

The allocations are performed on lines 7-12. The thunk closure is allocated at `Hp-48`. It consists of a code pointer (`sat_s00_entry`) and the free variables needed by the code. In this case, the free variables are the `to` argument stored in `R1` and the unboxed `from` integer stored in the stack. The new list element is allocated at `Hp-16`. It consists of a tag (`:_con_info`) and the head and tail of the list. The head of the list is the `from` element which is loaded from the stack, and the tail of the list is the thunk that was just allocated at `Hp-48`. After the allocations are done we are prepared to return to the caller of the `upto` function.

We load the return value into `R1` in line 13. The return value is the new list element we just allocated. It is tagged with a two in the low-order bits by pointing `R1` to `Hp-14` instead of `Hp-16`. The tag indicates that the closure is evaluated and contains a cons cell as opposed to an empty list (which would be tagged with a one). The stack space allocated by the `upto` function is deallocated in line 14. Finally, in line 15 we return to the caller of the `upto` function by jumping to the return address stored on top of the stack. When the thunk we just allocated is demanded later in the program's execution it will cause the control to jump to the `sat_s00_entry` code.

Figure 3.6 shows the code for the thunk allocated by the `upto` function. The code for the thunk will be entered when the value of list element that holds the thunk is

```

1 sNO_ret()
2 {
3     cPr:
4         Hp = Hp + 56;
5         if (Hp > I64[BaseReg + 144]) goto cPu;
6         if (%M0_S_Gt_W64(I64[Sp + 16], I64[R1 + 7])) goto cPw;
7         I64[Hp - 48] = sat_s00_entry;
8         I64[Hp - 32] = R1;
9         I64[Hp - 24] = I64[Sp + 16];
10        I64[Hp - 16] = :_con_info;
11        I64[Hp - 8] = I64[Sp + 8];
12        I64[Hp + 0] = Hp - 48;
13        R1 = Hp - 14;
14        Sp = Sp + 24;
15        jump (I64[Sp + 0]) ();
16    cPx: jump (I64[BaseReg - 16]) ();
17    cPu:
18        I64[BaseReg + 184] = 56;
19        goto cPx;
20    cPw:
21        R1 = []_closure + 1;
22        Sp = Sp + 24;
23        Hp = Hp - 56;
24        jump (I64[Sp + 0]) ();
25 }

```

Figure 3.5: Low-level code for upto continuation point number two. It checks the termination condition and returns either the empty list or the next list element.

demanded. At that point, the thunk is responsible for performing the recursive call to `upto` with the `from` parameter increase by one.

The thunk code begins by checking to see if there is enough free space available on the stack and heap. Next, on line 7 we push an update frame onto the stack along with a pointer to the thunk closure that is stored in the `R1` register. The update frame is responsible for overwriting the closure with (an indirection to) the value to which the closure evaluates. These updates are necessary to avoid repeated evaluation of shared closures and are a required part of an implementation of lazy evaluation. The update will be performed when we return from the `sNO` function with the newly allocated list value.

Line 9 contains the increment of the `from` variable. The increment is done on the unboxed value (`m#`) stored as a free variable in the closure. Lines 10-11 allocate the new boxed integer that will be passed as the new value of the `from` parameter. We are now ready to prepare for the recursive call to `to upto`. Lines 11-12 load the two parameters into the expected registers. The `R2` register gets the first parameter, which is the freshly allocated integer. The `R3` register gets the second parameter which is the original `to` value that is store in the closure. Finally, we can call the `upto` function on line 14.

In this section we took a detailed look at the low-level code generated for a single Haskell function. It is important to note that each of the four low-level entities described in this section will be seen as separate functions by the LLVM back end. It is the complexity of the low-level code, introduced by the managed environment, the support for first class functions, and laziness, that creates the fundamental opportunity for optimization that our work tries to exploit. In the next section we examine how control flows between multiple functions and the effects that lazy evaluation and type classes have on control flow.

```

1  sat_s00_entry()
2  {
3      cPa:
4          if ((Sp + -16) < SpLim) goto cPc;
5          Hp = Hp + 16;
6          if (Hp > I64[BaseReg + 144]) goto cPe;
7          I64[Sp - 16] = I64[stg_upd_frame_info];
8          I64[Sp - 8] = R1;
9          _sOL::I64 = I64[R1 + 24] + 1;
10         I64[Hp - 8] = GHC.Types.I#_con_info;
11         I64[Hp + 0] = _sOL::I64;
12         R2 = Hp - 7;
13         R3 = I64[R1 + 16];
14         Sp = Sp - 16;
15         jump Main.upto_entry ();
16     cPc: jump (I64[BaseReg - 16]) ();
17     cPe:
18         I64[BaseReg + 184] = 16;
19         goto cPc;
20 }

```

Figure 3.6: Low-level code for the upto thunk. It calls the upto function with an incremented from parameter.

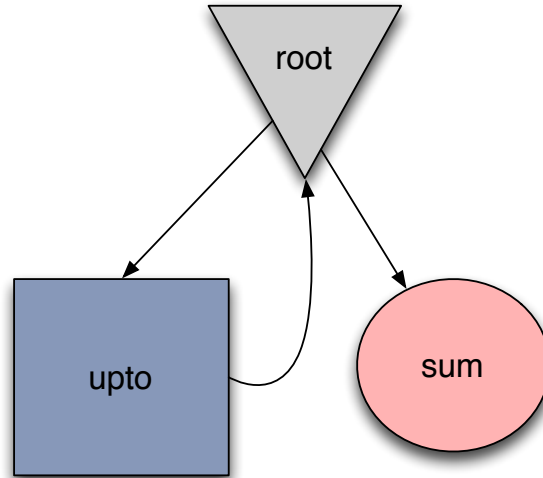


Figure 3.7: The non-lazy call graph with no type classes for the code in Figure 3.2.

3.3 Low-Level Control Flow

The program in Figure 3.2 is simple, but it demonstrates several of the overheads introduced by abstraction mechanisms in Haskell. In particular, we can see the overhead effects of lazy evaluation and type classes. These abstraction mechanisms interfere with the compiler’s ability to optimize the program because they obscure the control flow visible to the compiler. In this section we will examine the control flow in the example program and describe how it becomes obscured in the code generated by GHC. In this section we examine the control flow of the `sum` program and identify a simple optimization opportunity that becomes available when the control flow is exposed to the compiler.

Without lazy evaluation or type-class methods, we would expect the control flow to look like the image in Figure 3.7. The program begins in the `root` function which calls the `upto` function which generates the list of numbers and returns to `root`. Control flow then transfers to the `sum` function which computes the sum of all the numbers in the list. Although this description is accurate for an eager language, the actual control flow in the Haskell program will be quite different.

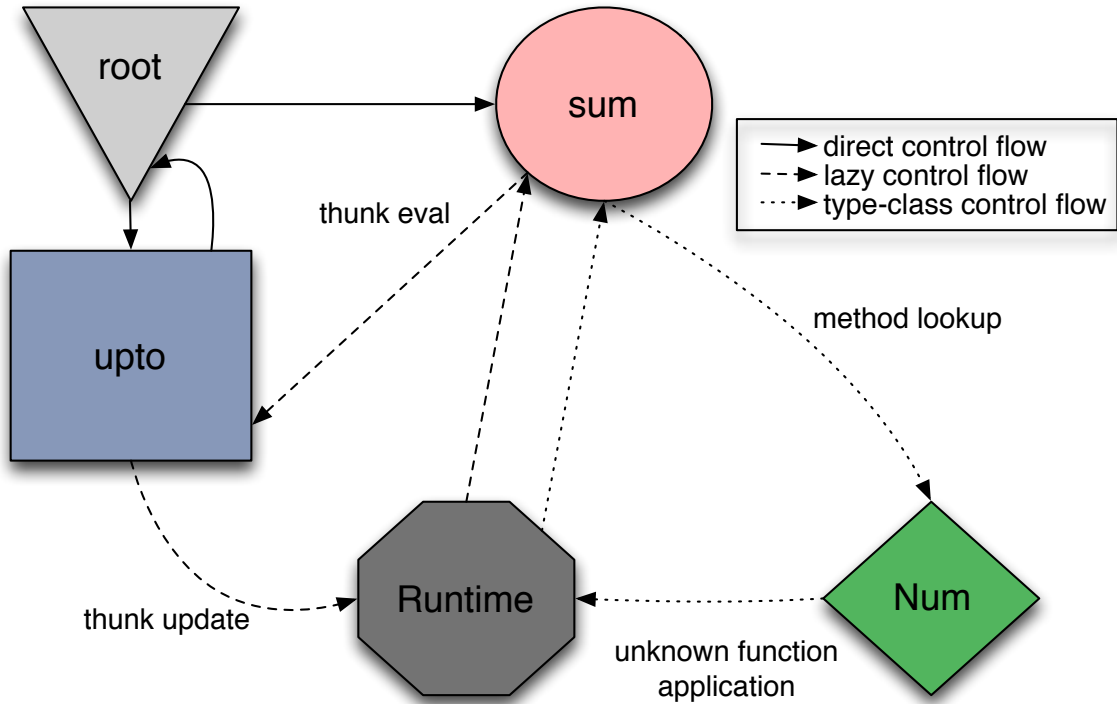


Figure 3.8: The lazy call graph with type classes for the code in Figure 3.2. The thick dashed lines are edges introduced from lazy evaluation and the thin dotted lines are edges introduced by type classes. The solid lines show the control flow as seen by the static compiler. The `root` function calls the `upto` function which returns a thunk to the root function. Next, the root function calls the `sum` function passing the thunk. The `sum` function will repeatedly evaluate the thunk to pull each number from the list as it accumulates the total.

Figure 3.8 shows the control flow that occurs in a language like Haskell with lazy evaluation and type classes. There are two new nodes in the control-flow graph and several new edges. The new nodes are for the `Num` type class and Haskell runtime. The dashed edges are for control flow induced by lazy evaluation and the dotted edges are for control flow induced by type-class method calls.

The path from the `sum` node to the `Num` node represents the call into the `Num` type class to look up the `+` function implementation and then use a generic application routine to apply the unknown (at compile time) function (see Marlow and Peyton Jones [2006] for details about generic application). The path from the `sum` node to the `upto` node is made by the lazy evaluation of the list of numbers. The list is represented by

a thunk that will generate elements of the list as they are needed. We took a detailed look at how that list is generated by the `upto` function in Section 3.2. When the list is inspected in the `sum'` function to see if it is empty or not, the actual control flow will shift back to the `upto` function to generate the next element. Before returning the element to the `sum` function, the thunk is overwritten with (an indirection to) the new list cell. The thunk must be updated in this way to avoid wasting execution time by repeatedly evaluating the thunk.

As we can see, the control flow for even this simple example is quite complex. Unfortunately, the actual control-flow graph gets even worse when we look at the real code generated by GHC. Figure 3.9 shows the control flow that was reconstructed from the actual executable code generated for the example program.

The primary addition to the control flow in Figure 3.9 is the new nodes for the return points from case evaluations. These evaluations will be done when the value of a function argument is needed. Since Haskell is a lazy language, there may be an arbitrary amount of code that is executed when inspecting a variable with a case statement (i.e. turning a thunk into a value may involve a lot of work). Because of this potentially unbounded amount of work, every time a variable is scrutinized with a case statement a continuation point must be created to hold the code necessary to continue the computation. Normally, these case continuations will be reached by an indirect jump after the evaluation of the case statement. However, with the addition of pointer tagging as described in Marlow et al. [2007] we can directly jump to the continuation point if the variable has previously been evaluated to a value.

Figure 3.9 also reveals the frequent execution paths in the program. There are two main traces in the example program. The first trace is the `upto` trace marked by the dashed blue line and corresponds to the evaluation of the list thunk. The second trace is the `sum` trace marked by the dotted red line; it corresponds to the `sum'` function that inspects the list and performs the addition with the accumulator. Combining the

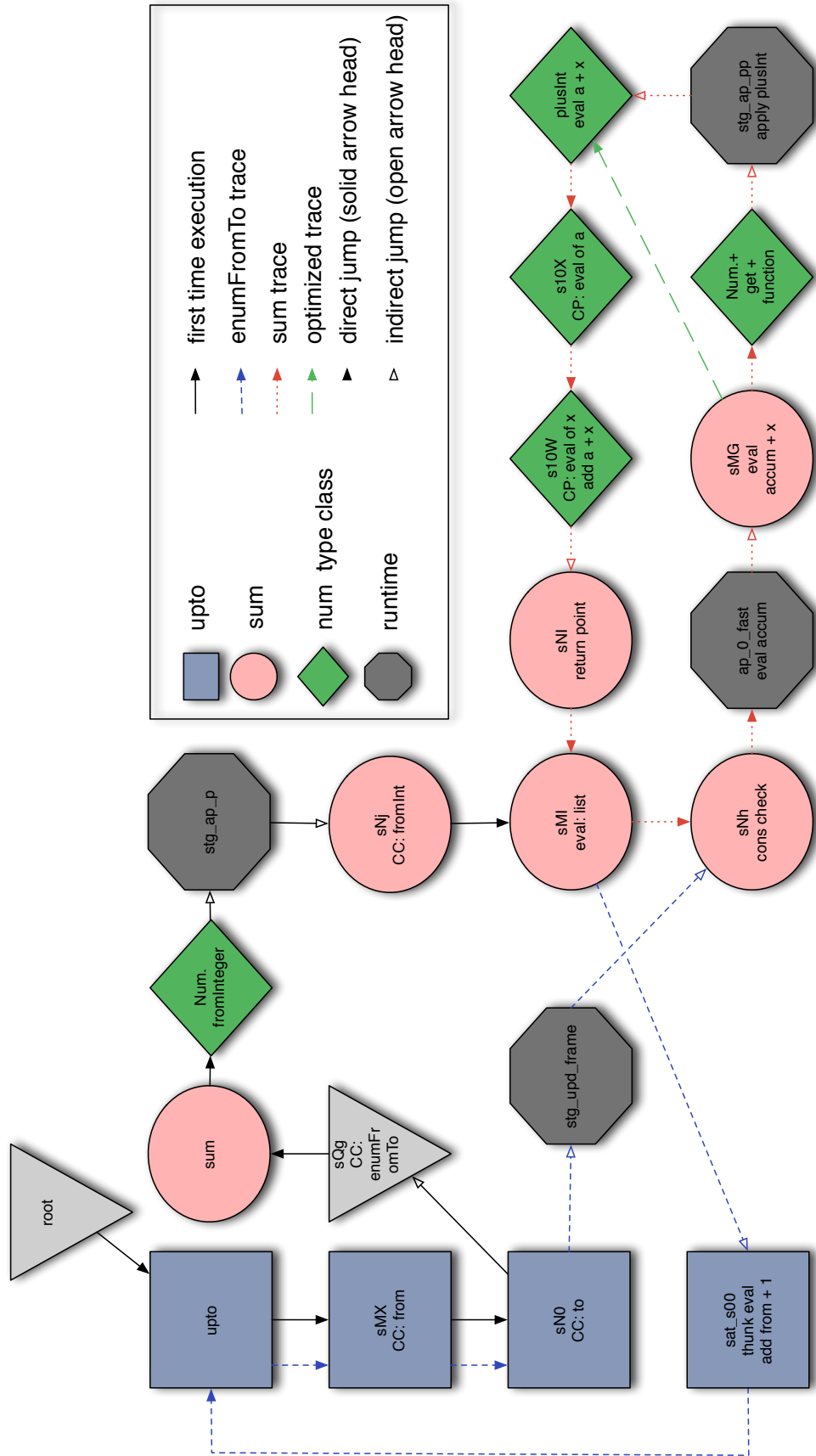


Figure 3.9: Hand-coded trace example for the program in Figure 3.2.

two traces results in the loop that consumes the majority of the program's execution time. As we can see, the frequent execution path tends to contain nodes from a variety of source level functions as well as runtime and `Num` type class functions. This mixture of nodes is only visible at runtime; the static compiler has no chance to reconstruct this flow at compile time.

Once the control flow is apparent, we can see an opportunity to reduce the overhead of type classes using a technique similar to the inline method cache used to efficiently implement Smalltalk as described by [Deutsch and Schiffman \[1984\]](#). We could modify the code so that the `+` function is not looked up in the type class dictionary every time through the loop. Instead, we place a direct call to the function stored in the dictionary. This optimization is effective because it eliminates the lookup of the function in the dictionary and the application of the unknown function returned by the dictionary lookup. Further, the optimization is best performed based on runtime information when we will know the value stored in the dictionary and can use that value as the target of the function call. The optimized trace is indicated in [Figure 3.9](#) by the long-dashed green arrow.

In this chapter we took a detailed look at the low-level code we get for a Haskell program and how lazy evaluation and type classes effect the control flow of a program. From our observations it would appear that Haskell programs have some unique low-level code shapes compared to traditional programming languages. These code shapes, in turn, may create opportunities for optimization that are not apparent in either the original Haskell code or in the low-level code generated by GHC. In [chapter 6](#), we will develop techniques to find and expose these opportunities by creating traces through the low-level code. In the next chapter we will measure a variety of Haskell programs and compare their low-level behavior to the behavior of C and C++ programs.

Chapter 4

Low-Level Haskell Behaviors

In this chapter we examine the low-level behavior of Haskell programs and compare them to programs from the SPEC benchmark suite. The goal of this study is to see if we can distinguish Haskell programs from programs written in traditional programming languages. These low-level behaviors give an insight into the code shape of the program. [Cooper and Torczon \[2012\]](#) define *code shape* as “all of the decisions, large and small, that the compiler writer makes about how to represent the computation in both IR and assembly code.”. Code shape is important because it dictates the optimization opportunities available to the compiler. We saw in the previous chapter how making the control flow manifest enabled an optimization for reducing the overhead of applying type-class functions. We are interested in applying the traditional compiler optimizations found in LLVM, so the code shape of Haskell will have a direct bearing on the effectiveness these optimizations.

4.1 Low-Level Behaviors

Measuring the exact code shape would account for both common sequences of instructions and common control-flow patterns in the control flow graph. Even in a small program, the number of combinations of instructions are large. We therefore

approximate the code shape by measuring four key properties: the mix of instructions executed, the different types of jumps executed, the number of targets for indirect branches, and the length of the basic blocks. The rationale for each of these measurements is discussed below.

The instruction¹ mix tells us both what instructions are generated and how much variety exists in the instruction stream. The identification of the instructions is useful for an analysis of what instructions are commonly executed by a running program. We can use this information to tell, for example, if a program performs many memory operations compared to the number of arithmetic and logic operations. Note that the instruction mix is not weighted by execution time so that simply performing more of a given kind of instruction does not mean that kind of instruction is responsible for more of the execution time of the program. The variety of instructions executed relates to the complexity of the translation to low-level instructions. A greater variety of instructions may indicate that a more complicated translation is used to produce the machine code, and a more complicated translation typically indicates that optimizations were applied during the translation process.

Measuring the different kinds of jumps gives an idea about how much code the compiler will typically be able to see. Indirect jumps and calls obscure control flow, limit the scope of optimization, and decrease the precision of an optimizing compiler's analysis of the code. On the other hand, direct branches are easily dealt with in the compiler since it can build a precise control flow graph that contains edges to the known target(s) of the branch.

The number of targets for indirect branches is important for measuring the locality of branches. We saw in Chapter 3 that lazy evaluation can cause Haskell programs to jump to arbitrary code when evaluating function arguments. These lazy-evaluation induced jumps will be indirect jumps in the final program. If these jumps typically

¹I will use the term instruction and opcode interchangeably.

have few targets then we could expect to get some benefit by specializing the code for the different jump targets. If the number of targets is very large then specializing the code is unlikely to help.

A basic block is a maximal sequence of straight-line code with a single entry and a single exit. Basic blocks are standard units inside compilers because they provide a known context for optimization: if one instruction in the block executes, they will all execute. Many traditional local compiler optimizations work over the basic blocks in a program. Small basic blocks indicate that there is little computation taking place between the control flow operations in the program. Large basic blocks provide good opportunities for optimizations. The distribution of the lengths of basic blocks give some insight into the average size of the nodes in the control flow graph, but not into the shape of the graph.

4.2 Measuring Low-Level Behaviors

The code behaviors were measured by examining the hardware instruction traces generated by running the program. The traces are collected using the Pin tool of [Luk et al. \[2005\]](#). Pin works by dynamically instrumenting a program binary to provide callbacks to user defined functions at various times during execution. As the program executes, Pin builds up execution traces for the dynamically executed paths in the program. The first time a new trace is executed, we get an analysis callback that allows us to insert instrumentation code along the trace. Subsequent executions of the trace will include any code added by the analysis pass.

To collect the instruction mix, jump mix, and basic block length data we break the trace into its component basic blocks. For each basic block we record the mix of instructions in the block and its length. We then insert an instrumentation callback that executes every time the basic block executes. Our instrumentation callback

increments the count of the number of times that block executes. When the program has completed, we can compute the counts for the actual instruction mix, jump mix, and basic block lengths based on the information we recorded for each basic block. The collection of indirect branch target data is a bit more complicated.

The collection of the indirect branch target data proceeds in three main phases: instrumentation, consolidation, and histogramming. The instrumentation phase uses Pin's buffering API to record some data at the location of each indirect branch. A program has three different kinds of indirect branches: jumps, calls, and returns. We normally may not think of a function return as an indirect branch, but it qualifies since a return will jump to an instruction whose address is stored in memory (i.e. the return address on the stack). At each indirect branch we record three pieces of data into a buffer provided by Pin. We record the kind of indirect branch (jump, call, or return) the address of the branch instruction, and the address of the branch target. Using Pin's buffer API allows for more efficient instrumentation because we are simply writing into a data buffer at each indirect branch and avoid the overhead of calling out to a function to record the data. Pin manages the size of the data buffer and when it is full Pin will call a user defined consolidation function to process the data.

The consolidation function takes the buffer records and groups data based on the address of the branch instruction (the actual address of the branch instruction, not the address of the target of the branch). For each unique indirect branch instruction address executed by the program we keep a `JumpRecord` that keeps track of all the targets the branch reaches. To process a full buffer we examine each entry in the buffer and lookup the `JumpRecord` for the branch address (creating a new record if needed). Each `JumpRecord` maintains a set of target addresses and a counter for the number of times the branch was executed. Once we find the appropriate record for the source address of the branch we add the target address of the branch to the set of targets

and increment the counter. The consolidation routine is called repeatedly during the program until execution completes. When the program has finished executing, we process all `JumpRecords` to build a histogram.

The histogram of `JumpRecords` bins the indirect branches in the program by the number of distinct targets reached through the branch. The histogram maps the number of distinct targets to the branches that have that number of targets. For example, we will group together all branches that have one target, two targets, and so on. We can then sum the total number of times the branches having that number of targets is executed. The sum will tell us how frequently branches with the given number of targets are executed by the program.

An important point to note about the data collection for Haskell programs is that we do not collect data when the program is executing inside the GHC runtime. For our purposes, we can consider the GHC runtime as consisting of two parts: a *CMM-runtime* containing hand-written CMM code that is used to implement parts of the Haskell execution model, and a *GC-runtime* written in C code that implements runtime services such as the garbage collector and lightweight threads. The CMM-runtime contains important Haskell features such as the implementation of updating a thunk with its value and applying unknown functions to their arguments. It would be good to include the data for when we are executing in this part of the runtime. On the other hand, we do not want to include data from the GC-runtime since it is never visible to the compiler, and the GC-runtime features for garbage collection and lightweight thread creation are not relevant to our study of low-level compiler optimization. Unfortunately, we were unable to separately collect numbers for the different parts of the runtime because they are included in the same shared library.

Our goal is to characterize the behavior of Haskell programs to get a better understanding of how they might respond to compiler optimizations. We exclude the runtime execution from the behavior of Haskell programs to gain a more accurate

picture of the shape of the low-level Haskell code. Leaving out data for the CMM-runtime means that we will not see the effects of several commonly called functions (e.g. the `thunk update` function). Our resulting data will not show the runtime functions that are frequently called from many different spots, but it will also exclude the GC-runtime functions that have no bearing on the goals of this thesis. While not a perfect solution, our choice to exclude runtime data means that we get a more accurate picture of what LLVM typically sees when it compiles the low-level Haskell code. Excluding the runtime data requires a simple change to the collection strategy described above.

To make sure that we only instrument the desired parts of a program we add a dynamic check when adding the instrumentation. The check examines the address where we are adding the instrumentation. If the address comes from the main executable program or a library we want to instrument then we add the instrumentation. Pin provides a way to map an address to the image that contains the address. To get the list of acceptable images we ran our benchmarks with Pin and recorded each image that was loaded during program execution. The list was then edited by hand to only include libraries containing Haskell code or libraries containing code for the SPEC benchmarks. Using this technique we are able to avoid adding instrumentation for the Haskell runtime and the C runtime library. Restricting the instrumentation in this way provides a more accurate picture of the code shape as it is seen by the compiler.

4.3 Results

4.3.1 Instruction Mix

Figure 4.1 shows the mix of instructions executed by each benchmark. The categories for instruction types are given in Table 4.1. We can see that the SPECfp programs

Type	Meaning	Example
ARITH	Arithmetic operations (integer & float)	add, lea, mulsd
CONTROL	Control flow operations	jne, call, jmp
DATA	Data movement operations	push, pop, mov
LOGIC	Logic and comparison operations	and, cmp, maxsd, test
OTHER	Miscellaneous operations	cvtsi2sd, nop

Table 4.1: Categories used for classifying instruction types.

contain a much higher ratio of arithmetic operations compared to both the Haskell and the SPECint programs. We would expect to see that the Haskell benchmarks from the Dph and Repa groups have a larger fraction of arithmetic operations compared to other Haskell programs since they are array-oriented codes. As we suspected, the Dph and Repa benchmarks do have a higher percent of arithmetic operations compared to the Hackage benchmarks, but they fall far short of the behavior of the SPECfp benchmarks.

The Hackage benchmarks have a very uniform look to the distribution of their opcodes. It is remarkable to see so little variation in breakdown of opcodes. Although they look similar to the SPECint benchmarks, the number of arithmetic operations is generally higher and lacks the variation found in the SPEC benchmarks. The great uniformity of arithmetic operations for the Haskell benchmarks is likely due to the heap and stack pointer manipulations which show up as simple addition and subtraction operations.

Compared to the SPEC benchmarks, the Hackage benchmarks seem to have slightly more data movement operations and slightly fewer logic operations. The difference is not that pronounced, but can probably be attributed to the frequent heap usage by Haskell programs.

Somewhat surprisingly there does not appear to be a huge difference between the distribution of opcodes for Haskell programs and SPECint programs. We can see a big difference between the SPECfp and Haskell programs, even for the numerically-oriented benchmarks from the Dph and Repa groups. These results suggest that

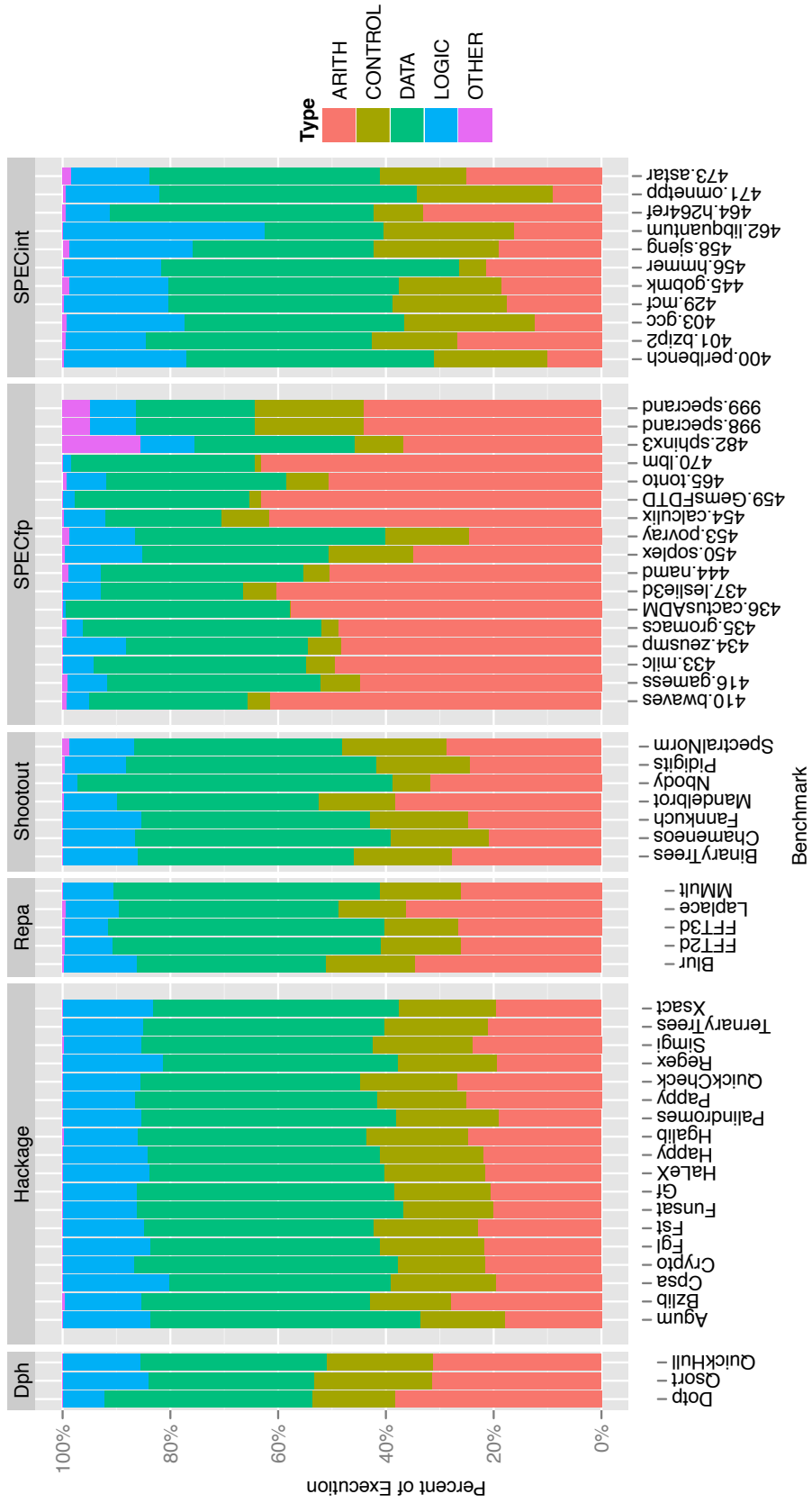


Figure 4.1: Mix of instruction types for Fibon and SPEC programs. The meaning of the instruction type is given in Table 4.1

Type	Meaning	Example
branch	Direct conditional or unconditional jump	<code>jne _L, jmp _L</code>
call	Direct function call	<code>call _foo</code>
indirect-branch	Indirect unconditional jump	<code>jmp %rax</code>
indirect-call	Indirect function call	<code>call %rax</code>
return	Return from a function call	<code>ret</code>

Table 4.2: Categories used for classifying branch types.

Haskell programs and C programs can expose similar code to the compiler and that optimizations that work well for the SPECint programs could work for Haskell programs as well.

These results show us properties of the overall executions of the benchmarks. Next we will zoom in on the control flow operations used by the programs to see if we can distinguish Fibon and SPEC programs based on the distribution of control flow operations.

4.3.2 Branch Mix

Figure 4.2 shows the distribution of branch types for programs from the Fibon and SPEC benchmark suites. Table 4.2 gives the meaning of the labels for the different types of branches. The first observation we can make is that standard, or direct, branch instructions are by far the most common type of branch executed across all the benchmarks.

The most striking observation we can make about this data is the difference in the number of indirect jumps between the program groups. With the exception of the Dph group, we can see that Haskell programs have a much larger number indirect branches than their imperative counterparts. There are two reasons for this difference. First, the implementation of lazy evaluation used by GHC will cause an indirect branch to be executed whenever an unevaluated parameter is needed in a function. Section 3.2 contains a detailed explanation of the indirect jumps needed



Figure 4.2: Mix of branch types for Fibon and SPEC programs. The meaning of the branch type is given in Table 4.2

to implement lazy evaluation. The second reason for the greater number of indirect jumps is that GHC manages its own Haskell call stack separated from the C call stack. A Haskell function calls another using a `jmp` instruction and when a Haskell function call needs to return, it does not use the `ret` instruction but rather executes an indirect jump to the return address.

The data we collected does not distinguish between indirect jumps due to lazy evaluation and those due to function returns. If we take the SPECint benchmarks as a model for the number of return instructions typically executed then it appears that Haskell programs require a significant number of indirect branches beyond those required to implement function returns.

Unlike the instruction mix in Section 4.3.1, there appears to be a significant difference in the types of branches executed by Haskell programs and those executed by SPEC programs. We can contribute this difference to lazy evaluation and the separately managed Haskell call stack. It is likely that this difference will be significant in terms of code shape. The large number of indirect branches make it difficult for the compiler to model the control flow. Without an accurate model of control flow the compiler is limited to optimizing over much small scopes which generally reduce the opportunities for optimization.

4.3.3 Indirect Branch Targets

Figure 4.3 shows the raw data collected for indirect branch targets. The graph plots a point for each group of indirect branches that have the number of targets indicated by the y-axis. We can see that the majority of the data is clustered near the bottom of the graph, which indicates that there are few branches with large numbers of targets. The graph does show that it is not uncommon to have indirect branches with a large number of targets. The SPECint benchmarks in particular seem to have benchmarks that contain indirect branches with a large number of targets. The large

number of targets is likely due to the presence of routines that get called from many distinct places. The returns from these functions are indirect branches that jump back to the many points from which they were called. The over plotting in this graph makes it slightly difficult to see all of the details. The next graph uses jittering and transparency to get a better impression of the data trends.

Figure 4.4 shows distribution of indirect branch targets broken down by the type of indirect branch. The plot has been rendered with the points jittered to better reveal data distribution. We can see a distinct difference in the types of indirect branches between Fibon and SPEC programs. The Haskell programs have many more jump branches compared to the SPEC programs which have many more return branches. This difference is attributed to the fact that GHC implements its own call stack, so that the jump instructions are actually used both for the implementation of lazy evaluation and for function call returns. We can again see that the majority of the branches are clustered in the part of the graph that indicates a fewer number of targets. These graphs have shown the distribution of branch targets without taking frequency of execution into account. We next examine the branch target data when it has been weighted by execution counts.

Figure 4.5 shows the distribution of indirect branch types weighted by execution frequency. As expected, the Fibon benchmarks show that nearly all of the indirect branches executed are from jumps. We expect to see this because return instructions are rare in GHC-compiled programs since GHC is managing its own call stack. The one outlier is Bzlib which uses C function calls through the Haskell Foreign Function Interface (FFI) to access libbz2 for data compression. Somewhat surprisingly, the SPEC programs are not totally dominated by return instructions. The inclusion of jump and call types in the distribution for the SPECfp benchmarks can be partially attributed to the fact that these benchmarks typically make very few calls as seen in Figure 4.2. Since they make so few calls, it does not take many indirect jumps in

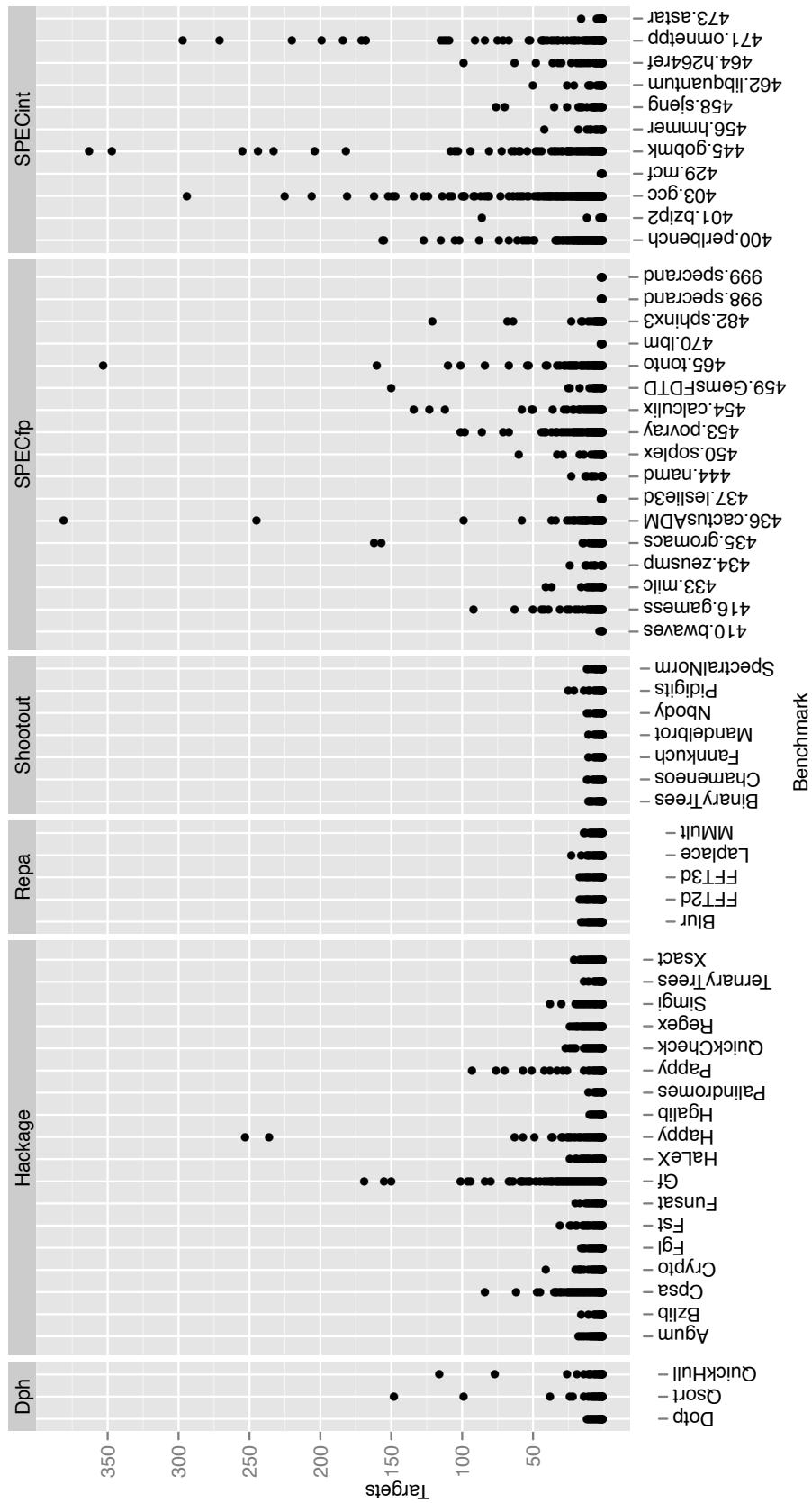


Figure 4.3: Number of targets for the indirect branches in Fibon and SPEC programs. We can see most of the data is clustered at the bottom of the graph, which indicates that most indirect branches have a small number of targets. However, we do have examples of branches with a large number of targets, particularly in the SPECint group. Over plotting makes it a little difficult to see all the features, but they are explored in later graphs.

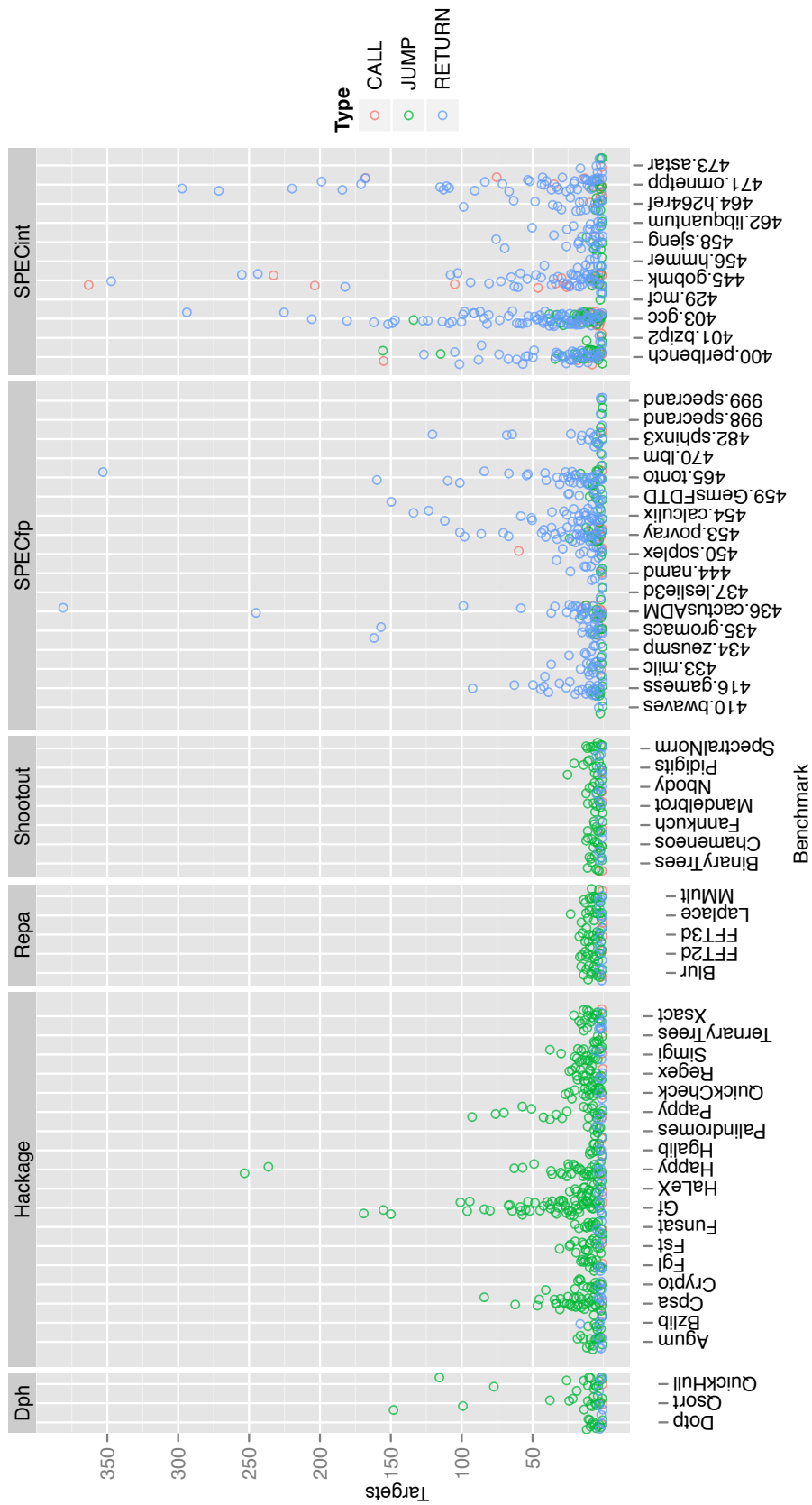


Figure 4.4: Number of targets for the indirect branches in Fibon and SPEC programs. The different indirect branch types are plotted separately in the graph. This graph uses jittering and transparency to better reveal where the bulk of the data points lie.

the program to dominate the distribution. Although most of the SPECint programs have return statements as their most frequent indirect branch type, there are some benchmarks such as `perlbench` that have a large portion of indirect calls and jumps. We could reasonably expect to see more indirect jumps and calls in the SPECint benchmarks compared to the SPECfp benchmarks because of their more control-oriented structure.

Figure 4.6 shows the weighted execution percent of indirect branches broken down by the number of branch targets. The first thing to notice is the large number of Fibon benchmarks that execute a lot of indirect branches that have only one target. These single target branches are likely due to lazy evaluation; they could either be from the indirect jump that happens when evaluating an unevaluated parameter or from the return jump to the continuation. In general many of the benchmarks appear to be execute a large number of branches that have only one or two targets. Overall the SPEC benchmarks seem to have more indirect branches that have more than two targets. The SPECint benchmarks in particular appear to have many indirect branches with multiple targets.

4.3.4 Basic Block Length

Figure 4.7 shows the raw data collected for basic block lengths. Each point in the graph represents that the benchmark contains a basic block of that size. The size of the point indicates the percent of the time that was spent executing blocks of that size. We can see from the data that the majority of benchmarks frequently execute basic blocks between 1 and 5 instructions in length. The one obvious outlier is `cactusADM` which executes blocks of length 70 or more 80% of the time.

Figure 4.8 shows the average basic block length for each benchmark. The average length is computed as a weighted average that uses the execution percent for each basic block size as the weights. This graph more clearly shows that the SPECfp

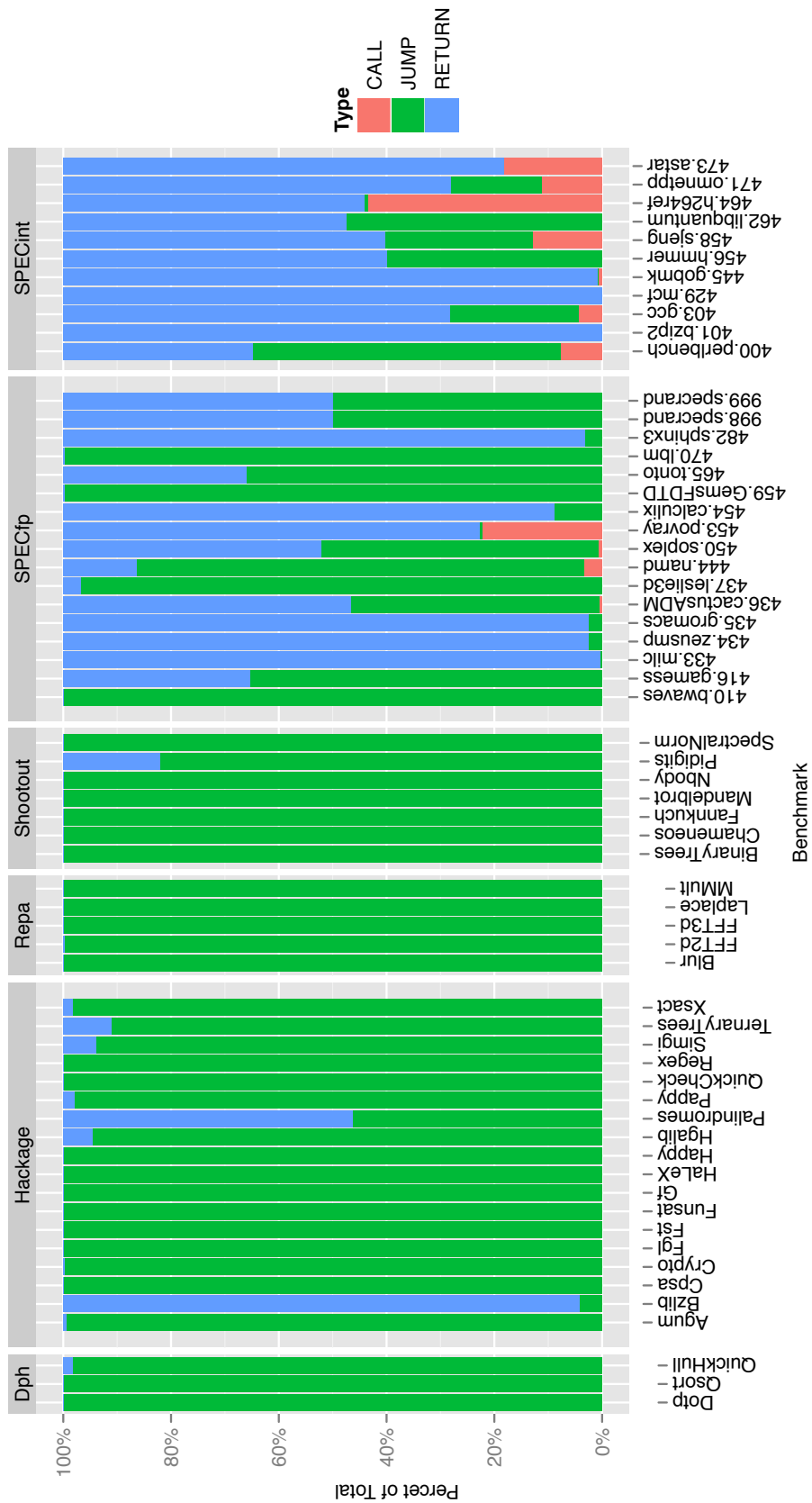


Figure 4.5: Indirect branch distribution by type for Fibon and SPEC programs. This graph shows the distribution of the types of indirect branches weighted by execution frequency.

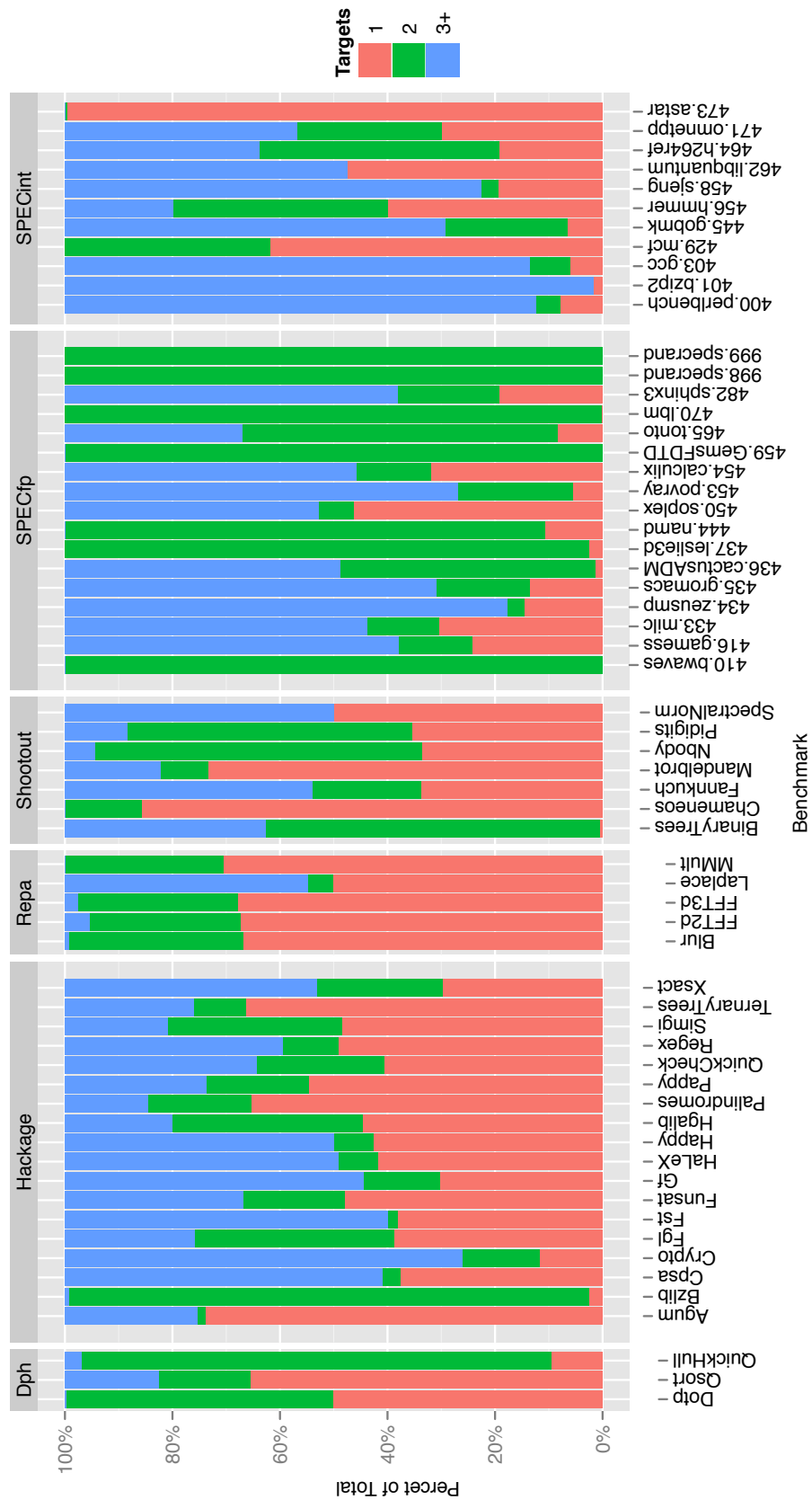


Figure 4.6: Indirect branch target execution percent for Fibon and SPEC programs. This graph shows the weighted execution percent of indirect branches broken down by the number of branch targets.

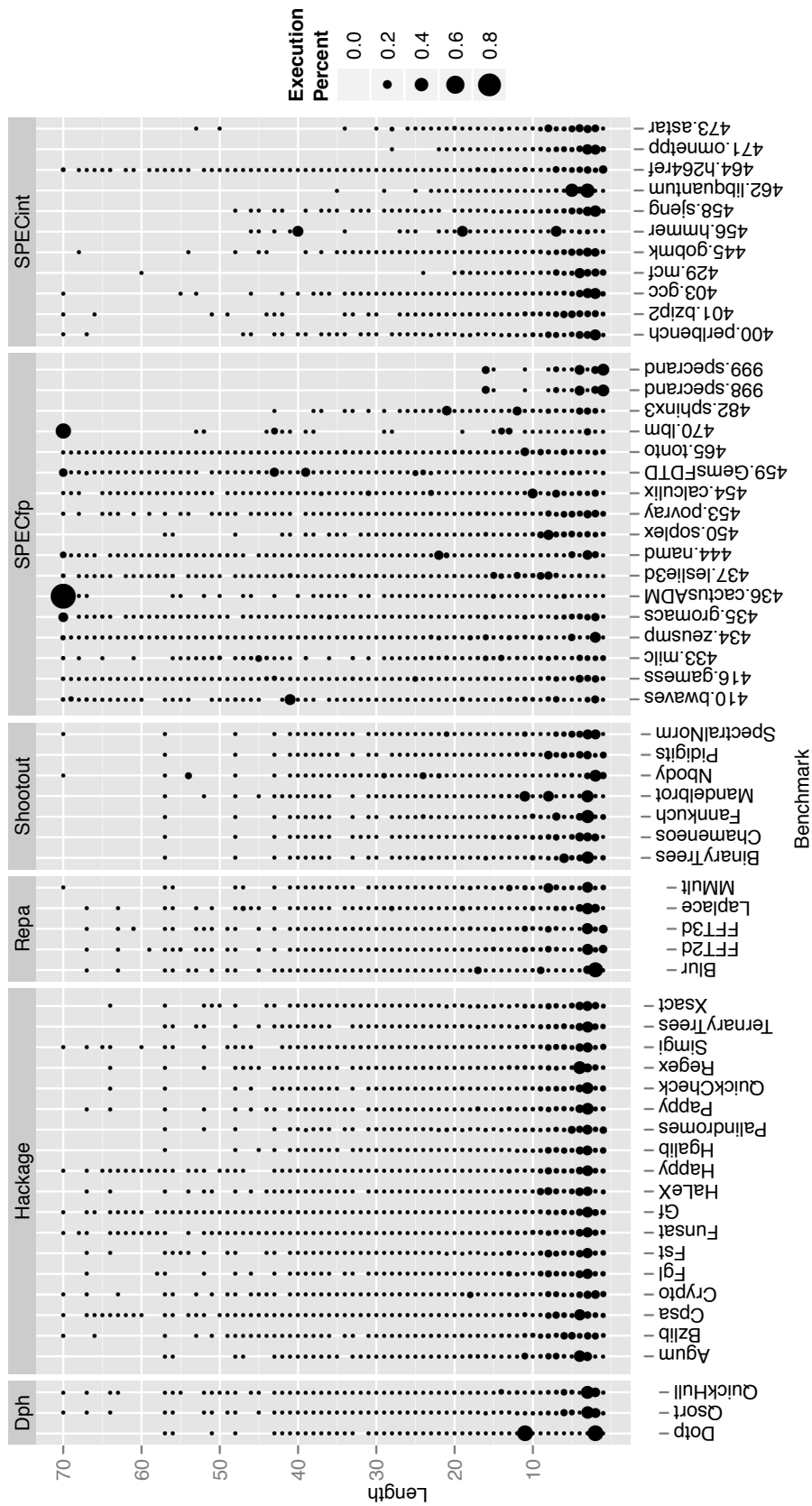


Figure 4.7: Mix of basic block lengths for Fibon and SPEC programs. The size of the point indicates the percent of the execution that was spent in blocks of that size. For example, the largest dot indicates that the program spent 80% of its execution time on blocks of that size.

benchmarks have a longer average basic block length than the rest of the benchmarks. We can zoom in on the remaining benchmarks by omitting the SPECfp group.

Figure 4.9 shows the average basic block lengths for the Fibon and SPECint benchmarks. We can see that the average lengths are all very similar at around 5-7 instructions per block. There does not appear to be a significant difference between the size of the basic blocks in Haskell programs and the SPECint programs. The SPECfp benchmarks do seem to have a longer average basic block length. This longer length could be because SPECfp programs spend a lot of time in loops that can be unrolled and optimized to increase the size of the blocks.

From the compiler’s perspective, the small basic blocks in Haskell programs means that we want to optimize over larger scopes. We are unlikely to find many optimization opportunities in these small basic blocks. Building an accurate model of control flow to enable global optimizations is likely to be important for optimizing Haskell programs since there will not be many local optimizations that will produce a big win.

4.4 Conclusion

In this chapter we measured the low-level behavior of Haskell programs from the Fibon benchmarks and compared them to C/C++/Fortran programs from the SPEC benchmark suite. We examined low-level behavior in terms of instruction mix, jump mix, indirect branch target counts, and basic block length.

The opcode mix revealed that Haskell and SPEC programs exhibit similar behavior in terms of the mix of instructions they execute. The jump mix shows that Haskell programs are unique in one regard: the number of indirect jumps executed. Haskell programs tend to have a much greater number of indirect jumps compared to SPEC programs. These jumps occur both because of lazy evaluation and because

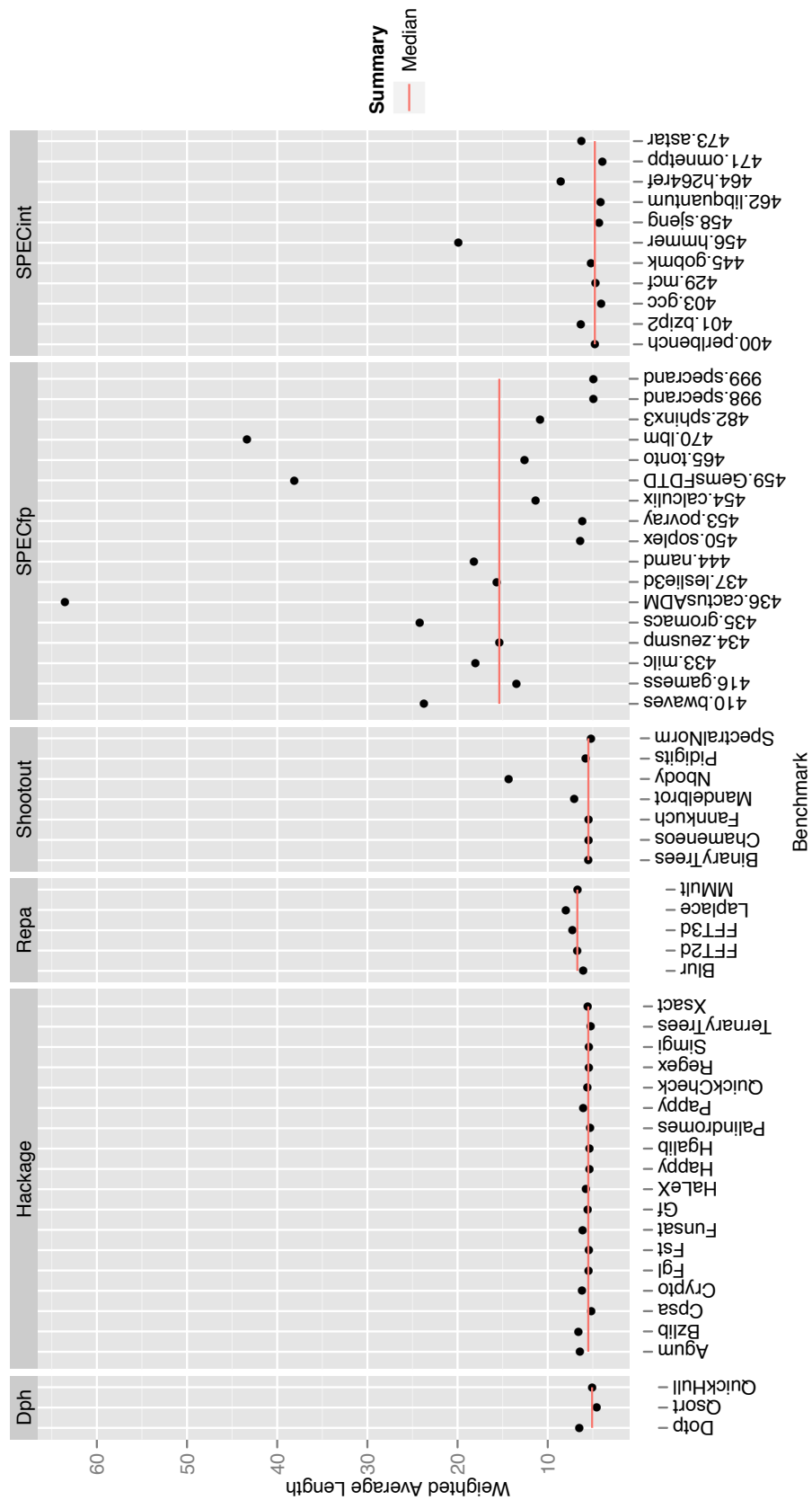


Figure 4.8: Average basic block lengths for Fibon and SPEC programs. The graph shows a weighted average for each benchmark, where the length of the blocks have been weighted by the execution percent for that size.

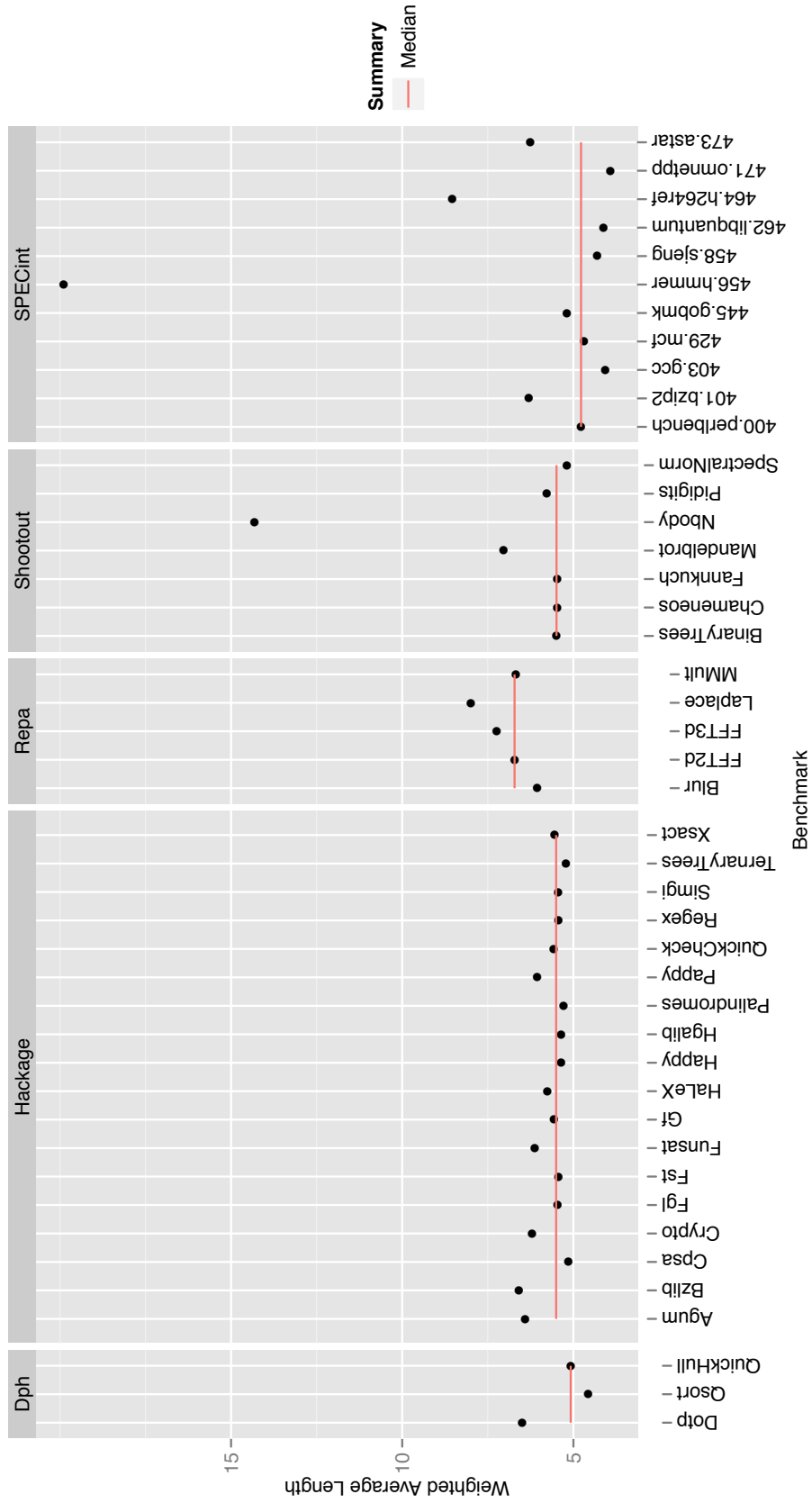


Figure 4.9: Average basic block lengths for Fibon and SPECint programs. The graph shows a weighted average for each benchmark, where the length of the blocks have been weighted by the execution percent for that size.

GHC manages the its own Haskell call stack separate from the C call stack. When we look at the number of targets for indirect branches in a program, we see that many of the indirect jumps in Haskell programs have only one or two targets. The small number of targets appears to be more pronounced in Haskell programs compared to SPEC programs. Finally, the length of basic blocks are similar for Haskell and the SPECint programs. Further, the average basic block length is quite short at around 6 instructions per block. This fact, in turn, may make the higher fraction of indirect jumps more significant because the precision of the control flow graph, or alternatively the exit from the CFG to another function, will make cross block optimization less effective.

The low-level behavior reveals some areas where Haskell differs from SPEC programs and shows the importance of optimizing over larger scopes. One way to increase the scope of optimization across blocks is to consider traces of frequently executed blocks. Since Haskell programs have many indirect jumps, these traces will need to be found at runtime so that the targets of the jumps can be made available to the compiler. The small number of targets for many of the indirect branches suggest that we can build stable traces by tracing through these indirect jumps. The next chapter examines our attempt to build program traces in Haskell programs using DynamoRIO, a binary trace-based optimization system.

Chapter 5

Dynamic Trace-Based Optimization of Haskell with DynamoRIO

In this chapter we look at a technique for building traces in Haskell programs by using a dynamic binary trace-based optimization system. As we saw in [Chapter 4](#), Haskell programs tend to have more indirect branches that obscure the control flow visible to the compiler than do programs in the SPEC benchmark suite. We suspect that the obscured control flow is one of the reasons that low-level optimizations are less effective for Haskell programs than for programs written in C, C++, or Fortran.

At runtime the targets of these branches are known and the control flow is clear. If we could build larger optimization scopes for the compiler by tracing through frequently executed paths then the compiler optimizations might be more effective. In this chapter we explore the impact of building program traces at runtime using DynamoRIO.

DynamoRIO [[Bruening, 2004](#)] is a system for manipulating programs at runtime by automatically building program traces and providing callback hooks for modifying

these traces. The traces are built as the program executes by monitoring the stream of instructions executed by the application. In this way, DynamoRIO works on an unmodified program binary without needing access to any source code. The program traces found by DynamoRIO extend seamlessly through both application and library code.

Each time DynamoRIO finds a new trace, it calls the callback functions installed by the user giving them a chance to optimize the traces. The traces are optimized in memory and the application continues executing by running the newly optimized trace. The original binary sitting on disk is never altered by DynamoRIO. Our initial idea for this chapter was to use the callbacks to optimize the traces found by DynamoRIO. However, once we measured the overhead introduced by simply finding the traces we abandoned DynamoRIO for the approach described in the next chapter. This chapter describes our initial effort of using DynamoRIO for optimizing Haskell traces.

We begin the chapter by looking at a case study where we build and optimize a program trace by hand. This study shows that we can achieve a speedup of 39% on the sum benchmark presented in Chapter 3. We next explore a general technique for building program traces by running Haskell programs under DynamoRIO. Unfortunately, the performance results with DynamoRIO are rather poor. The Fibon benchmarks show an average slowdown of 57% when running under DynamoRIO. We look at the root cause of the slowdown and discover that the main problems are poor trace formation and a bad interaction with the garbage collector.

This chapter is divided into four main parts. In Section 5.1 we explore building traces by hand for a single benchmark to provide some motivation for our trace-based optimization approach. The remaining sections focus on building traces with DynamoRIO. In Section 5.2 we describe the basic execution environment for applications running in the DynamoRIO framework. Next, in Section 5.3, we examine the

performance of Haskell and SPEC programs running through DynamoRIO without any optimizations. Finally, in Section 5.4, we present an analysis of the program traces found by DynamoRIO.

5.1 Hand-Coded Trace Case Study

In this section we present the results of a hand-coded case study of the performance potential of trace-based optimization for Haskell. The goal of this study is to show that trace-based optimization can improve the performance of Haskell code. The study examines a simple program and builds traces by hand for the common execution paths. We explore one transformation to build an optimized trace that reduces the overhead of type-class dictionaries. The initial results are encouraging. By running the trace just for the common execution path we can achieve a 15% speed improvement, and by running the optimized trace that reduces type-class overheads we see a 39% improvement. The results demonstrate that trace-based optimization has the potential to speed up the execution of Haskell programs.

The program used in the case study is the example program from Chapter 3 and is shown in Figure 3.2; it computes the sum of integers from 1 to 300,000,000.

To test the effectiveness of trace-based optimization, we experimented with building the traces by hand and performing a single optimization. The traces were hand written in assembly code using a combination of `ghc -O2 -S` to get the assembly output of the program and `gdb` to disassemble the machine code for the library and runtime functions used by the program. The snippets of assembly were collected together and the control-flow instructions were modified so that the fall through path stays on the trace. The remaining original code was left unmodified except for changing one control-flow instruction to jump to the trace at the appropriate point. The assembly code was then compiled and linked using `gcc` to produce an executable

program.

Results were collected for four different versions of the traces shown in Figure 3.9 on page 37. The first version only used a trace for the evaluation of the list thunk as indicated by the dashed blue line. The second version used the trace for the evaluation of the sum function as indicated by the dotted red line. The third version linked the two traces together by placing a jump at the end of the `upto` trace to the beginning of the `sum` trace. A final version of the trace used an optimized version of the linked trace and shows the limit of how good our inter-procedural knowledge can be for this program.

As described in Section 3.3, the trace was optimized using a technique similar to the inline method cache used to efficiently implement Smalltalk as described by Deutsch and Schiffman [1984]. We modified the assembly code so that the `+` function is not looked up in the type class dictionary every time through the loop. Instead, we place a direct call to the function stored in the dictionary. This optimization is effective because it eliminates the lookup of the function in the dictionary and the application of the unknown function returned by the dictionary lookup. Further, this optimization is an example of an opportunity we would expect to find in a trace-based optimization system. The optimized trace is indicated in Figure 3.9 by the long-dashed green arrow.

Table 5.1 shows the performance results for the trace experiment. Each version was run 100 times and the execution time compared to a baseline execution that contained no traces. The program was optimized by GHC at level `-02`. The speedups reported here are the median execution time of the original program divided by the median execution time of the hand-built trace version.

The results show that the `upto` trace alone improves the performance by 5% over the baseline version. The `sum` trace improves the performance by 8%, and linking the traces improves the performance by 16.0%. The linked trace performance is better

Trace	Mean	Median	StdDev	Median Speedup
None (baseline)	6.91	6.83	0.32	1.00
EnumFromTo	6.58	6.50	0.25	1.05
Sum	6.24	6.34	0.33	1.08
Linked	5.81	5.89	0.24	1.16
Optimized	4.94	4.92	0.09	1.39
None (inlining)	4.38	4.37	0.06	1.56

Table 5.1: Hand-coded trace performance. The table reports the runtime of the baseline program compiled by GHC at `-O2` to the runtime of the various trace versions. The Mean and Median columns report the execution time in seconds. The Speedup column is the median baseline time divided by the median traced-version time.

than the total of the two individual traces, a result which indicates that combining traces can produce better performance than the sum of the individual improvements. The optimized trace performs the best with an overall improvement of 39%, suggesting that such an optimization could be profitable in programs that make heavy use of type class methods.

As mentioned in Section 3.1, we stopped GHC from inlining the `sum` and `upto` functions. If we allow GHC to inline those functions, it can achieve an improvement of 56.5%. This result shows that GHC finds additional optimization opportunities that are enabled by inlining. However, the performance improvements we get from building and optimizing traces are still encouraging because GHC will not always inline functions in a program and we would expect to find control flow in real code similar to what we have seen in this example program.

In this section we saw that by creating and optimizing traces through the commonly executed portions of a program we can achieve a performance improvement of up to 39%. The results indicate that trace-based optimization can be profitable for Haskell programs. A major challenge to address is how to build these traces automatically without incurring more overhead than benefit. In the next section, we will discuss our results for building traces with DynamoRIO, which is a state-of-the-art runtime optimization framework.

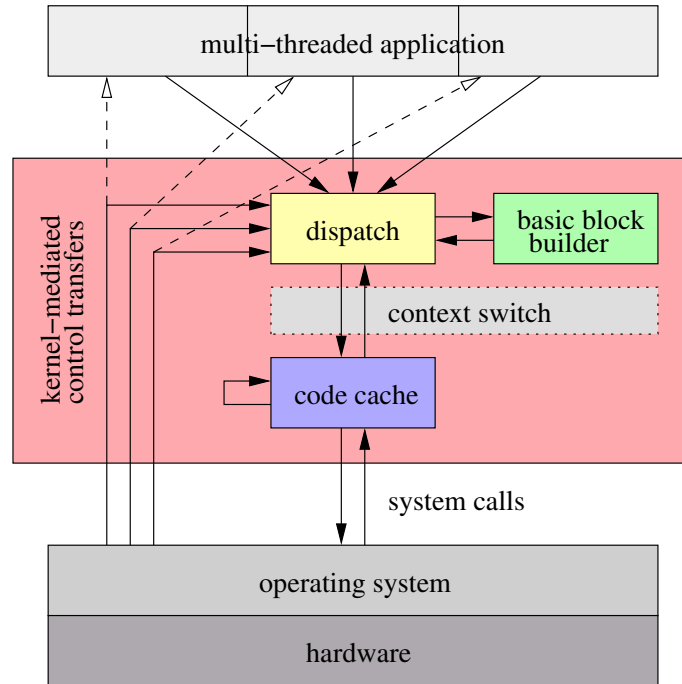


Figure 5.1: Overview of DynamoRIO. This is Figure 1.3 from [Bruening \[2004\]](#)

5.2 How DynamoRIO Works

In this section we describe the DynamoRIO framework for dynamic optimization. This description is based on the thesis of [Bruening \[2004\]](#), which is the most comprehensive description available. We will present the most important features of DynamoRIO as they pertain to our work.

Figure 5.1 shows an overview of DynamoRIO. When an application runs under the control of DynamoRIO, its code is not executed directly. Instead, the code is run out of the *code cache* and DynamoRIO copies the application code into the code cache as it executes. The first time a basic block is executed by the application it is copied into the code cache. The code is then directly executed from the code cache and control is transferred back to DynamoRIO so that it can find the next basic block to execute.

While the execution scheme described above is simple, it is not very efficient.

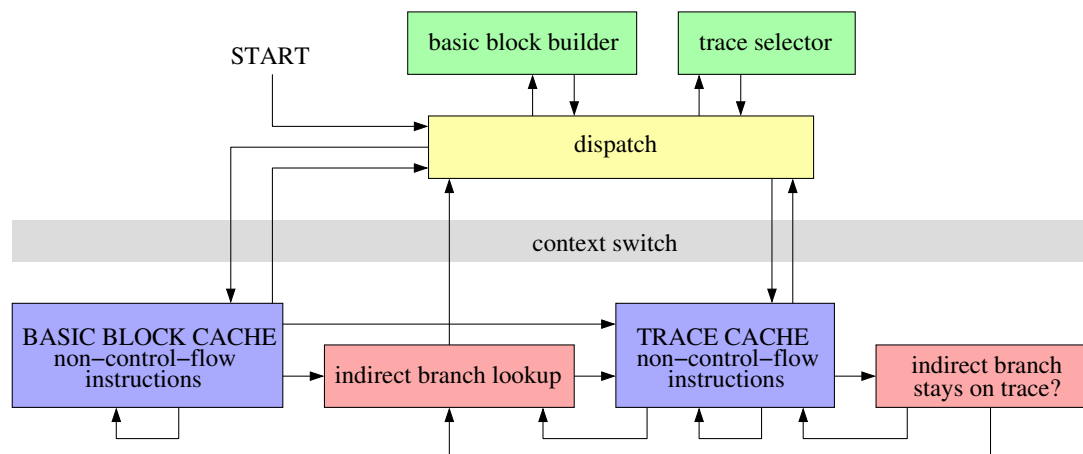


Figure 5.2: Code Cache Overview of DynamoRIO. This is Figure 2.1 from Bruening [2004]

DynamoRIO contains a few features to improve the efficiency of applications running under its control. The main goal is to keep the program executing in the code cache as long as possible to avoid the costly context switch needed when we must lookup the next basic block to execute. The two primary mechanisms to keep programs executing in the code cache are exit-stub linking and trace building. Figure 5.2 shows a more detailed picture of the code cache illustrating these features.

Exit-stub linking allows code to stay in the code cache by linking together basic blocks in the code cache. If a basic block ends in a direct jump and the target of the jump is already in the code cache, then the jump can target the other block in the code cache rather than returning control to DynamoRIO. Linking blocks together reduces the overheads of running through DynamoRIO because more time is spent in the code cache, which contains the actual code for the application. Exit-stub linking works well for direct branches because we know the target of the branch just by inspecting the code. Indirect branches are more difficult because the target of the branch depends on the data, and so the target of the branch can change over the execution of the program. DynamoRIO must perform a lookup of the indirect branch

target each time.

To look up the target of an indirect branch, DynamoRIO uses a hash table that maps addresses in the application’s address space to a trace or basic block in the code cache. The performance of the indirect branch lookup routine is important for applications that contain a lot of indirect branches. As we will see in the next section, the large number of indirect branches in Haskell are detrimental to the performance of DynamoRIO because of the time they spend in the lookup routine.

One way that DynamoRIO reduces the overhead of indirect branches is by building traces of frequently executed paths in the program. Traces are a collection of basic blocks that correspond to an execution path in the application. DynamoRIO builds traces to reduce the overheads of running the application and to provide a context for optimization. To build traces, DynamoRIO uses a variant of the Next Executing Tail (NET) algorithm that was used in the original Dynamo system of [Bala et al. \[2000\]](#)¹. It works by keeping counters associated with potential trace heads. Trace heads are all blocks that are the targets of backward branches. When the counter on a trace head exceeds a certain threshold DynamoRIO enters trace-building mode and records the next execution of the blocks that are visited starting from that trace head until it reaches another trace head or a size limit. The trace is now stored in the code cache and branches that target the trace head are changed to point to the new trace.

An important optimization for reducing the overhead of indirect branches is performed when building traces. As DynamoRIO is building the trace it will inline the targets of indirect branches into the trace. Because the target of the indirect branch may change the next time it is executed, DynamoRIO will include a check to make sure that the target of the indirect branch is the same as when the trace was originally created. If the target has changed, then the trace is exited and control returns to Dy-

¹NET was called MRET (Most Recently Executed Tail) in the original Dynamo paper, but renamed to NET in the subsequent paper by [Duesterwald and Bala \[2000\]](#)

namoRIO to perform a lookup for the target of the indirect branch. This optimization is very similar to the inline method caches of [Deutsch and Schiffman \[1984\]](#).

In the next section we look at the performance characteristics of programs running under the control of DynamoRIO.

5.3 Performance of Applications with DynamoRIO

Applications running under DynamoRIO will suffer a performance penalty for the time they are running DynamoRIO code instead of application code. All of the time spent outside the code cache is pure overhead. To get an understanding of how much overhead we incur from running under DynamoRIO we looked at the performance of the Fibon and SPEC benchmarks. These results show mostly the overhead of DynamoRIO without all of the potential benefits because no optimizations are performed on the traces. The one benefit we do see is from the straightening and compacting of the code into traces.

Figure 5.3 shows the performance penalty for the Fibon benchmarks running under DynamoRIO. We see that the benchmarks suffer an average slowdown of 57%. The Hackage benchmarks suffer the worst with an average slowdown of over $1.75\times$. Figure 5.4 shows the performance of the SPEC benchmarks under DynamoRIO. Compared to the Fibon benchmarks, the SPEC benchmarks suffer much less of a performance hit with an average slowdown of 12%. Although no optimizations are performed on the traces, the overhead of DynamoRIO looks quite high. We next tried to determine the source of these overheads using program counter (PC) profiling.

DynamoRIO provides a mechanism to repeatedly sample the program counter during execution and record what was executing when the sample occurred. Using this technique we are able to get an idea of how much time is spent in the various parts of DynamoRIO. Figures 5.5 and 5.6 show the profiling results for the Fibon

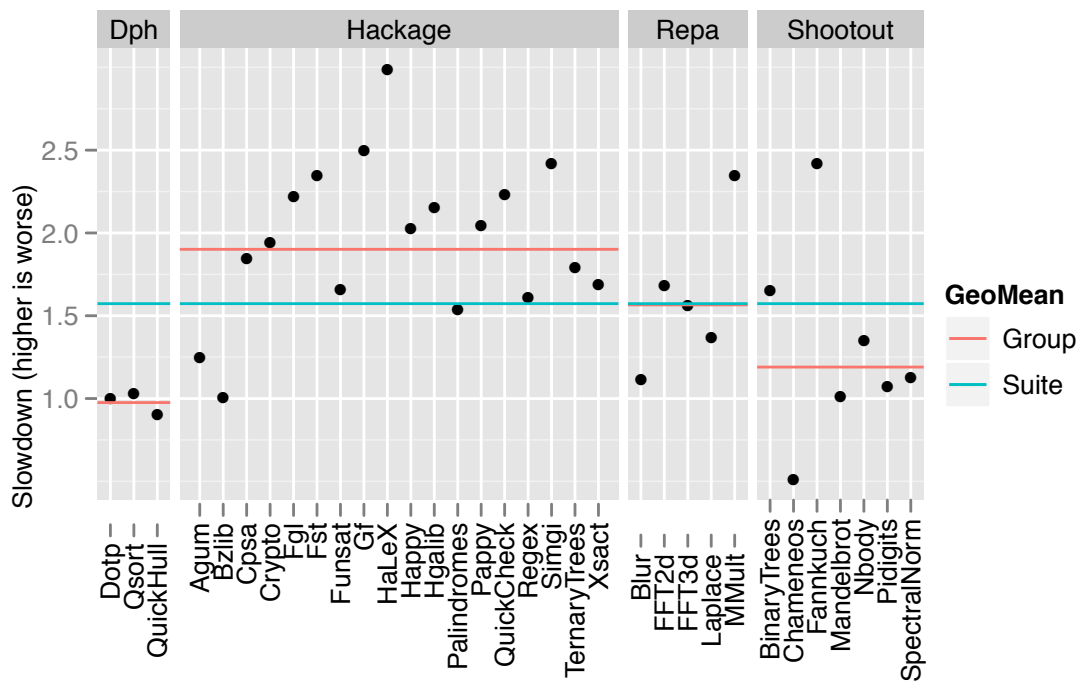


Figure 5.3: Performance of Fibon benchmarks under DynamoRIO. The benchmarks suffer an average slowdown of about 57%.

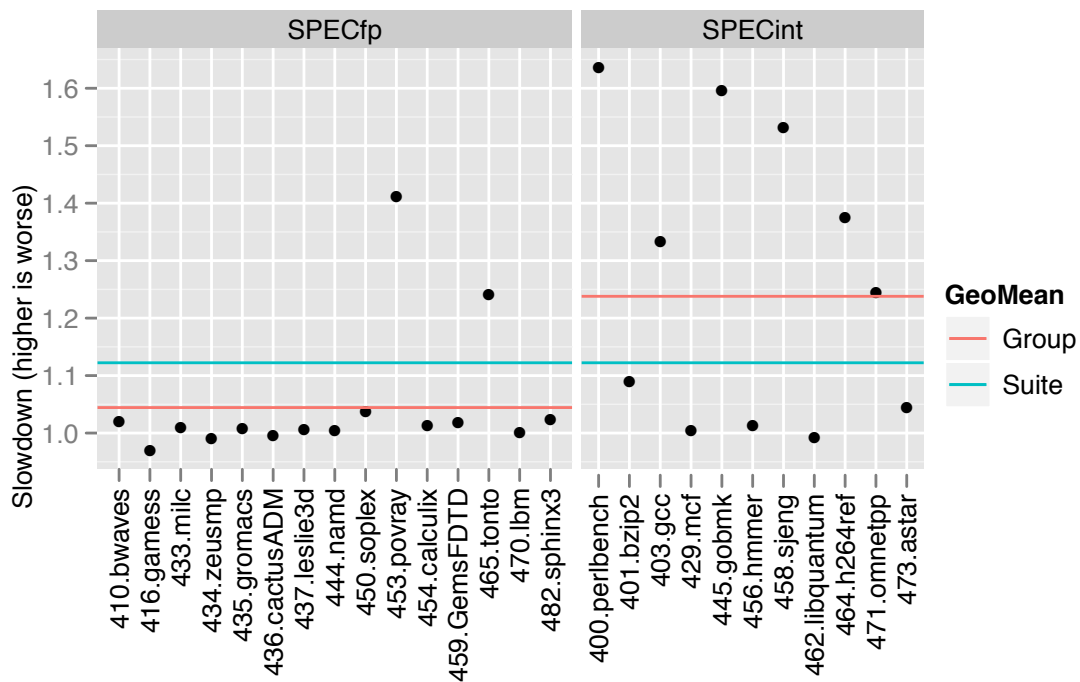


Figure 5.4: Performance of SPEC benchmarks under DynamoRIO. The benchmarks suffer an average slowdown of about 12%.

Location	Meaning
Interp	Translating basic blocks to code cache
Dispatch	Control flow outside of code cache
Monitor	Trace head counter monitoring
Syscall_Handler	System calls
IBL	Indirect Branch Lookup
Off.Trace	In the code cache not on a trace
In.Trace	In the code cache on a trace
Unknown	Some other location

Table 5.2: Key for PC profiling results graphs in Figures 5.5 and 5.6.

and SPEC benchmarks. Unfortunately, some of the Fibon benchmarks failed to run properly when the PC profiling was enabled so we have a reduced set of data for these results. A key for the meaning of the sample locations is given in Table 5.2

The profiling results reveal several interesting observations. First we can see a big difference in the distribution of PC samples between the Fibon and SPEC benchmarks. The prominent difference in the PC samples is that the Fibon benchmarks spend much more time in the indirect branch lookup (IBL) routine. We can attribute this difference to the large number of indirect branches in Haskell programs. We saw in Section 4.3.3 that a fair number of the indirect branches have only a single target. Apparently, the traces found by DynamoRIO are not able to exploit the branch target locality and it has to fall back to the full lookup in the indirect branch hash table.

Although many of the Fibon benchmarks spend a great deal of time in the IBL routine, there are some benchmarks that spend relatively little time there. The Pidigits, Nbody, Mandelbrot, and Bzlib benchmarks spend a reduced amount of time in the IBL compared to other Fibon benchmarks. The Nbody and Mandelbrot benchmarks spend most of their execution inside small loops, so DynamoRIO has no problems finding those traces. The Pidigits and Bzlib benchmarks spend time in external libraries so their code does not have the same indirect-branch characteristics as the other Haskell benchmarks. The Pidigits benchmark spends time in the libgmp li-

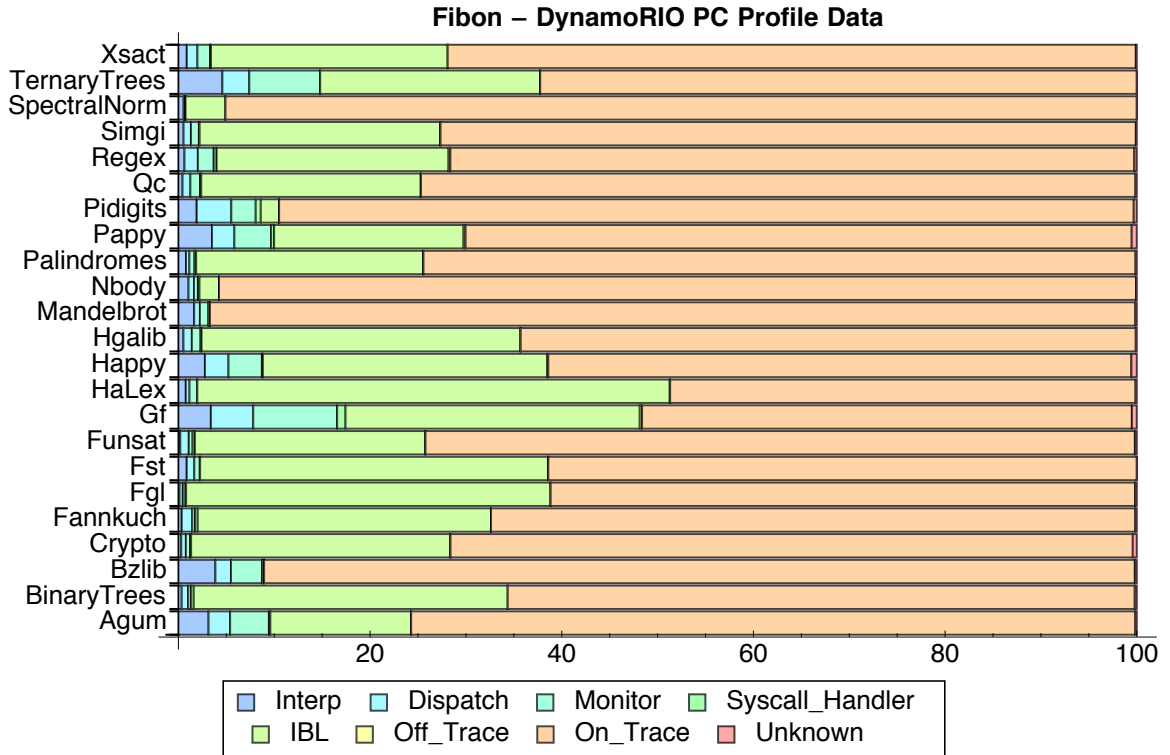


Figure 5.5: PC profile results for Fibon benchmarks under DynamoRIO.

rary performing multi-precision arithmetic and the Bzlib benchmark spends its time in the libz2 library.

The SPECint benchmarks in general spend more time in the IBL than their SPECfp counterparts. The primary exceptions are the mcf, hmmer, libquantum, and astar benchmarks. We can see in Figure 4.6 that these are the benchmarks that have a higher proportion of indirect branches that have a single target. The traces that DynamoRIO finds are able to exploit the single-target locality for these benchmarks. The IBL behavior of the SPECfp benchmarks are slightly harder to correlate with the indirect-branch data from Chapter 4. One obvious connection is the povray benchmark which is the benchmark that spends the longest amount of time in the IBL among the SPECfp benchmarks. As we see in Figure 4.5, the povray benchmark is unique among the SPECfp benchmarks in that it about 20% of the indirect branches that it executes are indirect calls. These calls are causing trouble for the

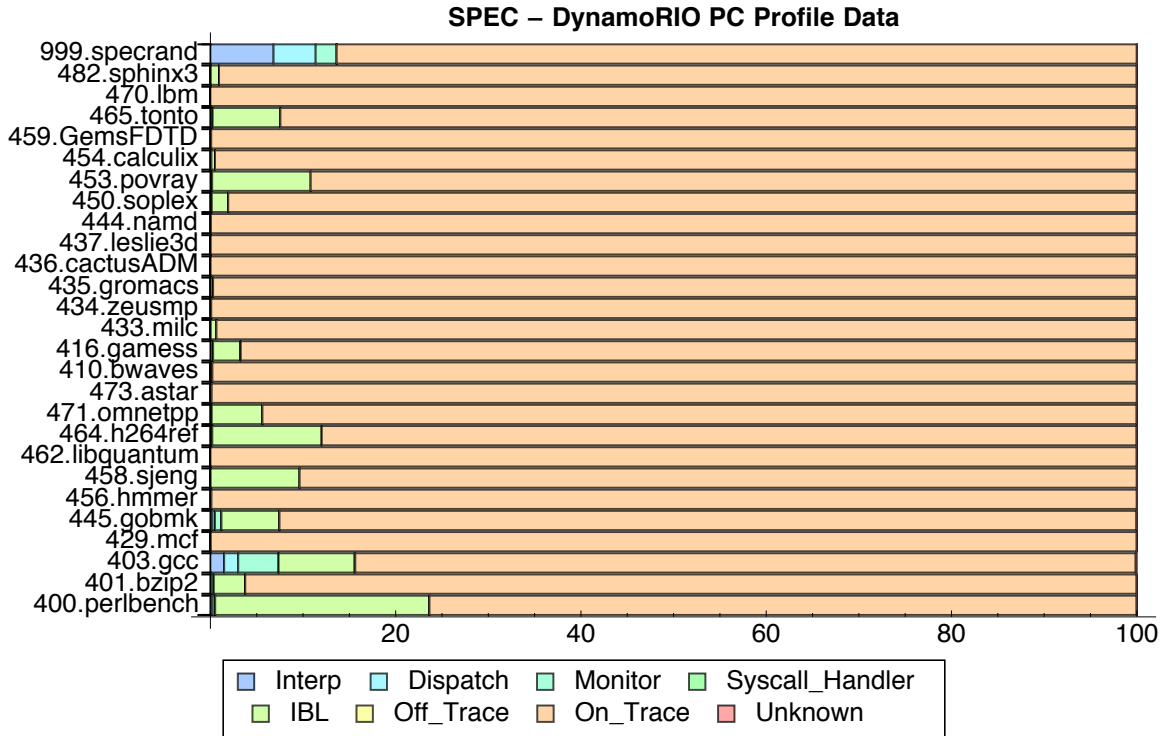


Figure 5.6: PC profile results for SPEC benchmarks under DynamoRIO.

trace heuristics used by DynamoRIO resulting in a large portion of the execution time to be spent in the IBL. It appears that DynamoRIO’s trace building heuristics have a sweet spot for SPEC behavior and a blind spot that diminishes their effectiveness on languages that induce higher levels of complex control flow such as Haskell or highly object-oriented languages like Smalltalk.

Looking at the DynamoRIO profiling results together with the performance results it looks like there is a strong correlation between time spent in the IBL and the magnitude of the slowdown. We can see that the Fibon benchmarks generally spend a lot of time in the IBL and their average performance under DynamoRIO is much worse than the SPEC benchmarks. Also, we can see that the two SPECfp benchmarks that perform the worst (povray and tonto) are the same benchmarks that spend the most time in the IBL routine. The correlation between slowdown and IBL time suggests that the IBL is a large component of the overall slowdown and is a major

cause of the slowdown we see. It would also suggest that the trace building heuristics are not generally applicable to all languages, but are rather tuned to the loop oriented codes from the SPECfp benchmarks.

The second interesting feature that we can see in the PC profile results is the observation that most benchmarks spend their code cache time executing traces and not single basic blocks. Both the Haskell and SPEC programs are similar in this regard. Once the overheads of DynamoRIO are accounted for, it appears that the majority of the time is spent on program traces as opposed to single blocks in the code cache. It is promising to see that traces get a good amount of execution time because it is traces that we want to optimize to improve the speed of the programs. Since we will want to optimize these traces, we need to get a better idea about the properties of these traces and how many traces we see in a typical program. The next section describes our efforts to answer these questions.

5.4 DynamoRIO Program Traces

We looked at two main properties of traces. First, we wanted to measure the length of an average trace. The longer the trace the more opportunities should exist for optimization since there is a larger context that can reveal inefficiencies. Also, the length of the trace contributes to optimization overhead since longer traces take more time to optimize. Second, we wanted to look at how frequently each trace was executed. If there are only a few traces that account for a majority of the execution time then we can potentially have a big impact by successfully optimizing the patterns that appear in a few traces. If the execution time is spread among many traces, then we might see less benefit from optimization since the cost of the runtime optimization must be amortized over the cumulative improvement in runtime from executions of the optimized trace.

The first trace property we looked at was the average length of the traces generated by DynamoRIO. Trace length is an important property because traces will be our unit of optimization. Presumably, longer traces contain more optimization opportunities. The length of a trace is measured in the number of instructions on the trace. We computed the average length using an average weighted by the frequency of trace execution. That is, the average trace length for a benchmark is computed as

$$Avg = \frac{\sum_{i=1}^n w_i x_i}{\sum_{i=1}^n w_i}$$

where x_i is the length of each trace in the benchmark, w_i is the number of PC samples falling on the trace, and n is the total number of traces. Using a weighted average gives us a better idea of the average length of traces that are executed the most frequently.

Figures 5.7 and 5.8 show the average trace lengths for the Fibon and SPEC benchmarks. We can see that the Fibon benchmarks have an average length of 42 instructions compared to 74 for the SPEC benchmarks. It is unclear whether 42 instructions is enough of a context to be able to perform significant optimizations, but it is conceivable that we can find optimizations over a scope of this size. Had the average length been very small, say around 10 instructions, it would call into question the trace-based optimization approach. One encouraging aspect of the trace length is that it is much higher than the average size of basic blocks we measured in Section 4.3.4. If our goal is to increase the optimization scope, then it appears that program traces are one way to achieve that goal. Now that we know the average length of the traces we will look at how execution time is distributed among the various traces found by DynamoRIO.

Figures 5.9 and 5.10 shows how much of the total execution time is spent on the most frequently executed traces in the Fibon and SPEC benchmarks. The results here are for only the single most frequently executed trace. The amount of time spent on

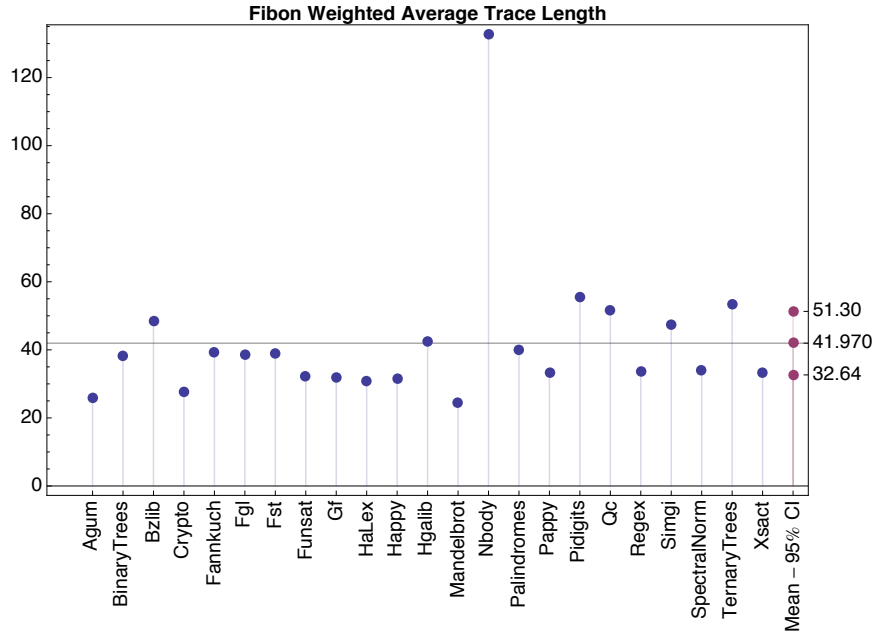


Figure 5.7: Average trace length for Fibon benchmarks. The average is shown together with a 95% confidence interval.

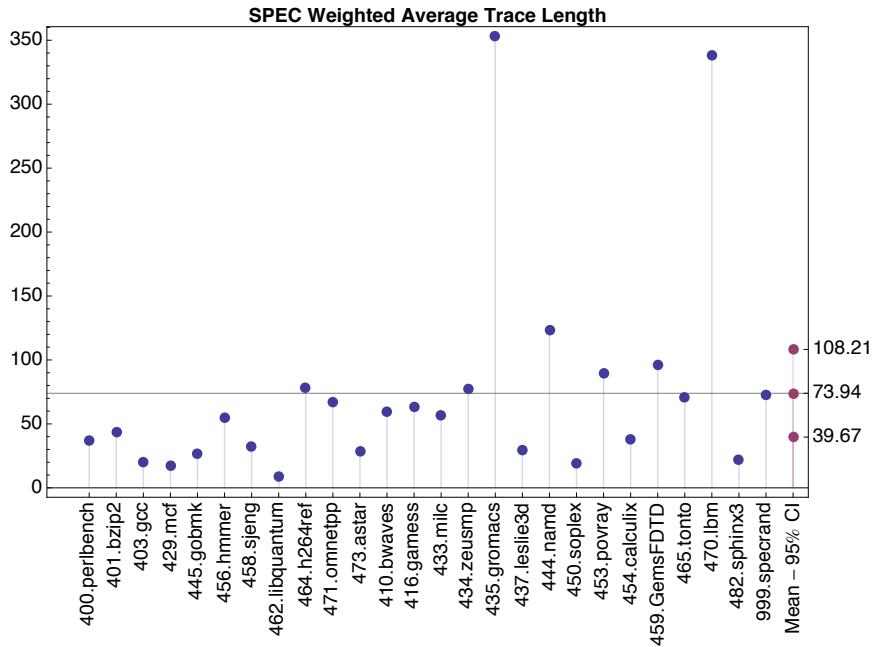


Figure 5.8: Average trace length for SPEC benchmarks. The average is shown together with a 95% confidence interval. Not shown is 436.cactusADM, which has an average length of 1815.83.

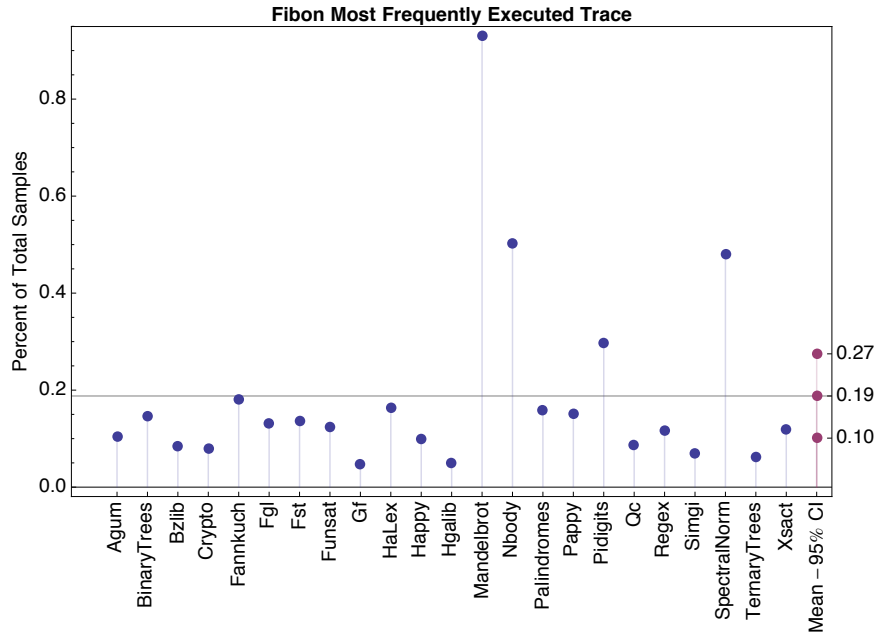


Figure 5.9: Execution percent of the most frequently executed Fibon traces. The average is shown together with a 95% confidence interval.

the most frequently executed trace provides a limit for how much speedup we could get by optimizing a single program trace. If the system could perfectly optimize it so that all instructions were eliminated then we could expect a speedup equal to the time that we spend on the trace. The results show that the Fibon benchmarks spend an average of 19% of their execution time on the most frequently executed trace compared to 24% for the SPEC benchmarks. These results are encouraging because they tell us that if we can successfully optimize the most important trace then we should see a real performance improvement. We next look at how the execution time is distributed among traces if we include more than just the most frequently executed trace.

In Figures 5.11 and 5.12 we look at how many different traces we need to account for 50% of the applications execution time. The summary statistic used in these results is the median number because there are a few outliers that throw off the arithmetic mean. As we can see, the Fibon benchmarks need a median of 12 traces

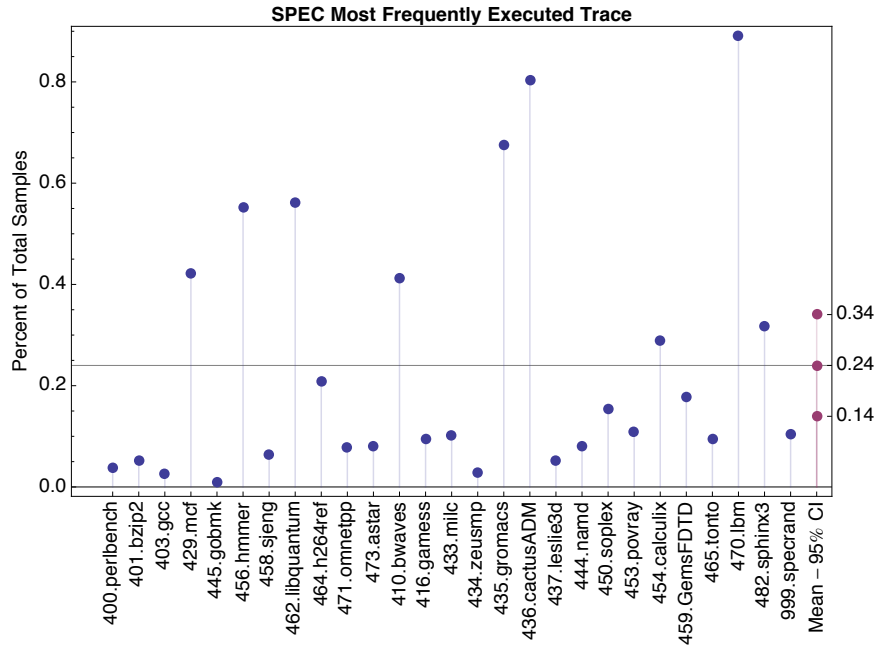


Figure 5.10: Execution percent of the most frequently executed SPEC traces. The average is shown together with a 95% confidence interval.

and the SPEC benchmarks need a median of 9 traces to account for 50% of the total number of PC samples taken. These results reinforce the idea that if we can find optimization opportunities in the small number of important traces then we should be able to reduce the running time of the benchmarks.

Although it takes few traces to encompass 50% of the trace samples, the overall number of traces found by DynamoRIO is quite large. The quartiles for the number of traces found by DynamoRIO is shown in Table 5.3. These results indicate that only 25% of the Fibon benchmarks contain fewer than 334 traces. This number of traces seems quite large compared to the small number of traces that actually contain most of the samples. To see where the large number of traces come from we took a detailed look at the traces from the sum program in Figure 3.2.

Figure 5.13 shows the low-level code from the sum program annotated with the traces found by DynamoRIO. There are two interesting features to point out in this figure. First, we can see that DynamoRIO has not identified the single long trace we

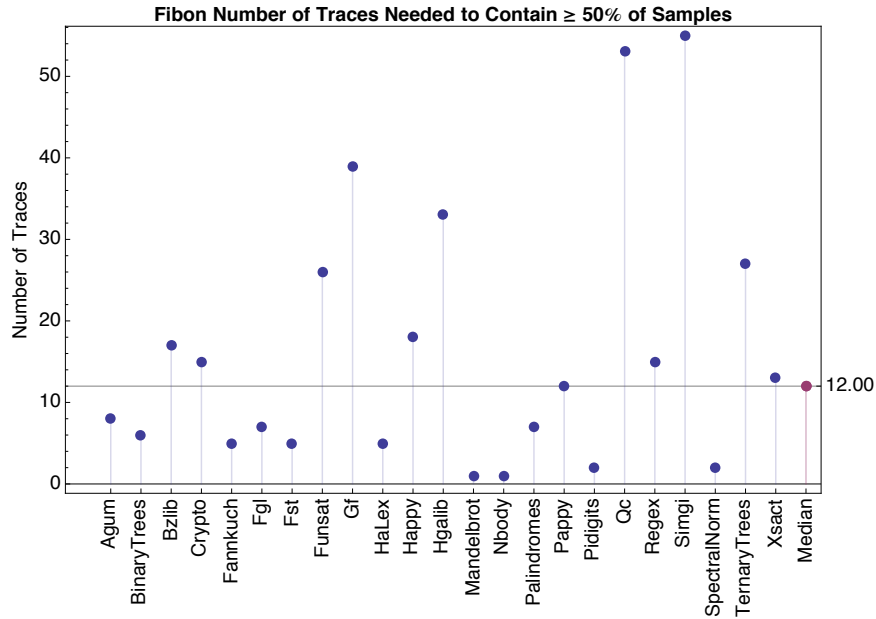


Figure 5.11: Median number of Fibon traces needed to encompass 50% of execution time.

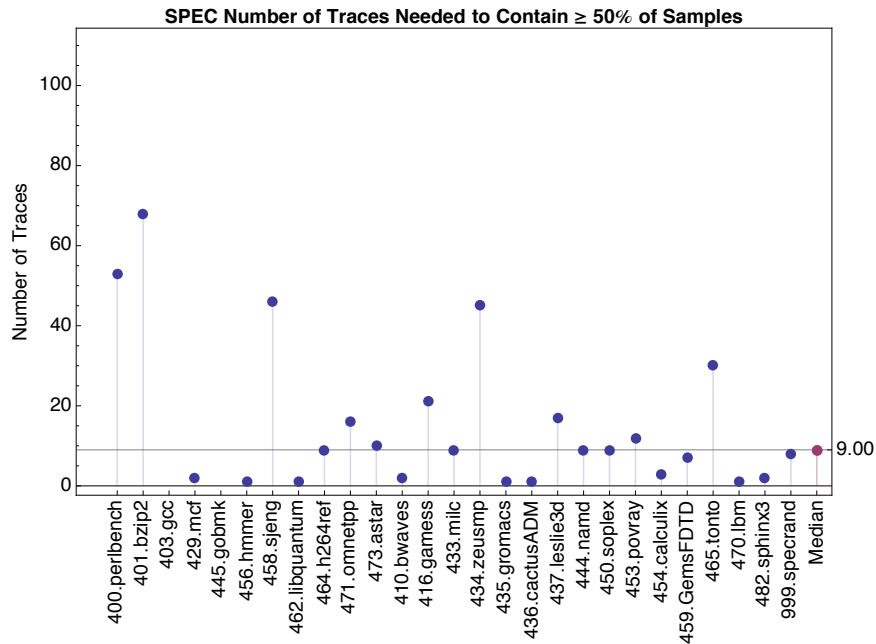


Figure 5.12: Median number of SPEC traces needed to encompass 50% of execution time. Not shown is 403.gcc, which requires 204 traces and 445.gobmk, which requires 614.

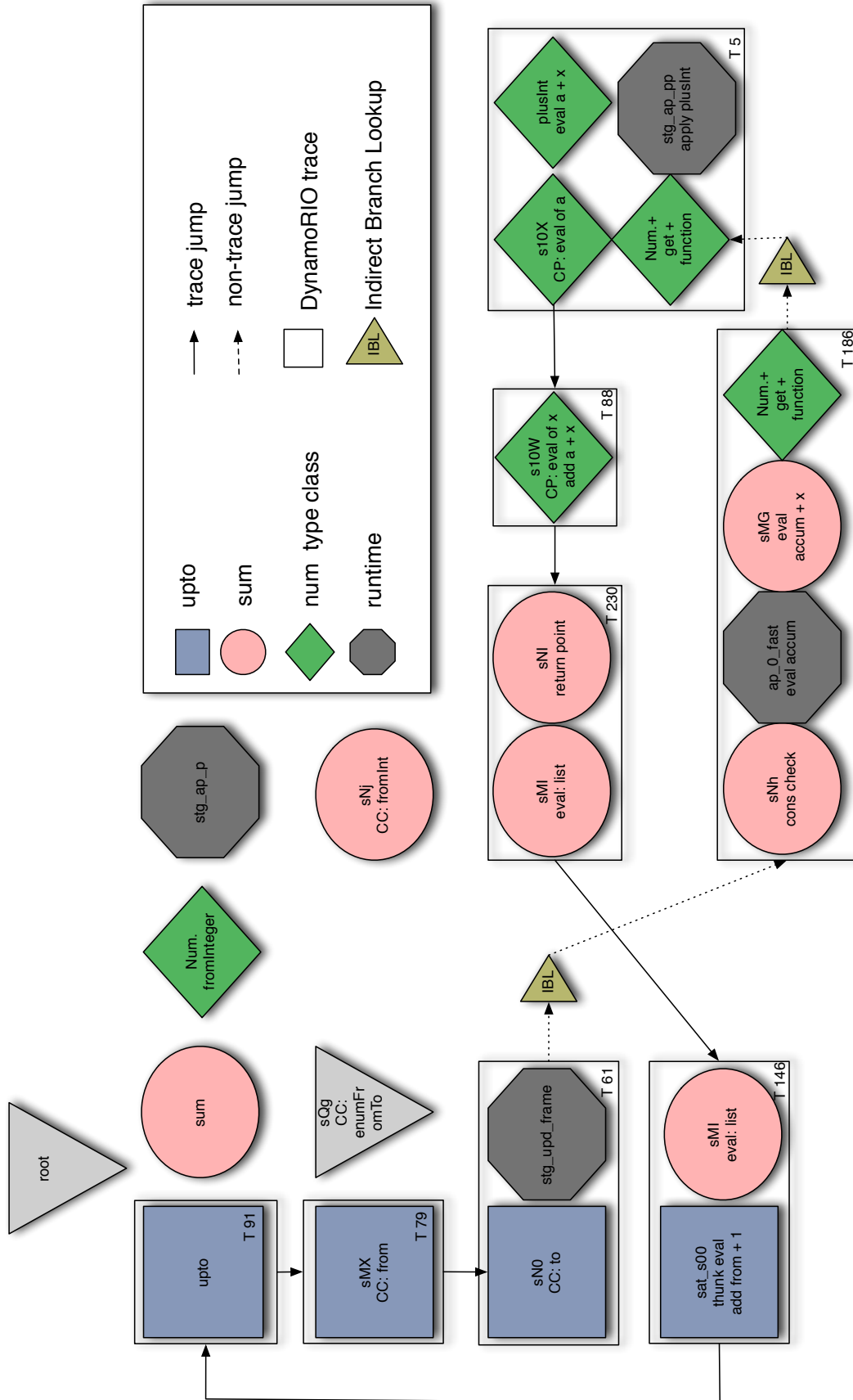


Figure 5.13: Sum program from Figure 3.2 annotated with the traces found by DynamoRIO.

Quartile	Fibon	SPEC
Q_1	334	220
$Q_2 = \text{Median}$	634	453
Q_3	1110	1446

Table 5.3: Number of traces found by DynamoRIO. The median number of traces (e.g. Q_2) found is 634 for the Fibon benchmarks and 453 for the SPEC benchmarks.

would like. Instead, the sum trace is broken into eight different pieces. Second, the traces are not all connected by direct jumps. Two of the traces end with calls to the indirect branch lookup routine. The failure of DynamoRIO to find the natural trace in this program is indicative of the performance problems that we encounter when using it to run Haskell programs.

If we look at the collection of all traces found by DynamoRIO for the sum program, we can see another problem with using a binary trace-based optimizer. Figure 5.14 shows a graph of all the traces found by DynamoRIO for our example program. The details will be difficult to see, but the general trend should be apparent. The graph has a node for every trace built by DynamoRIO. There is an edge between two traces if there is a direct jump (i.e. not going through the IBL routine) between the traces. We have colored the trace nodes according to how many PC samples fell on the trace. Blue traces had the fewest samples, followed by green, yellow, orange and finally red. Any trace that is not colored was never sampled by the profiler.

We can see that the sum trace is a tiny fraction of all the traces found by DynamoRIO. A feature that jumps out immediately is the large collection of nodes near the bottom right of the graph. These traces represent different paths through the garbage collector. This graph reveals one of the issues with building traces with a binary optimizer. The traces may include arbitrary parts of the language runtime that we do not want to trace.

In Haskell, we saw traces as a way to overcome limitations of the execution model, but building traces in a garbage collector is not necessarily a good idea because the

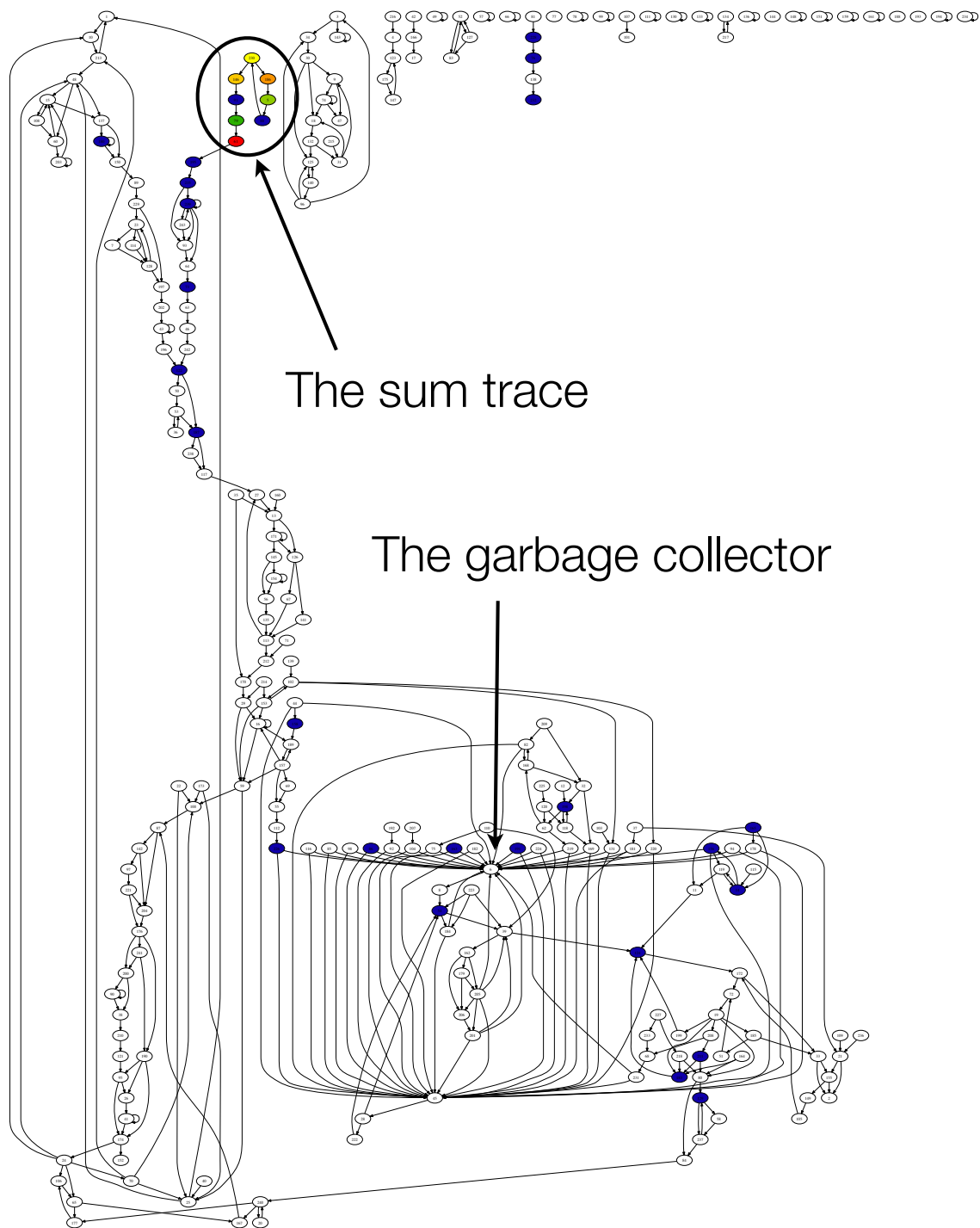


Figure 5.14: Graph of the traces found by DynamoRIO for the sum program in Figure 3.2. Each node is a separate trace and an edge between the nodes indicate there is a direct jump from one trace to another. Nodes without direct jumps between them are only reached by going through the Indirect Branch Lookup (IBL) routine. A filled node indicates that the PC profiling results had a sample fall on that trace.

execution path will be highly data dependent. These traces may not be a big problem if the program does not spend a large amount of time in the garbage collector, but they still add extra overhead that is undesirable from an optimization perspective. It is possible that the traces in the garbage collector causes Haskell programs to spend more time in the IBL routine due to bad traces through the collector. We attempted to correlate the efficiency of a benchmark (time spent outside of the collector) with the slowdown in DynamoRIO, but did not see any obvious connection.

We briefly tried to separate the Haskell runtime traces from the mutator traces without much success. DynamoRIO does have an option to start and stop the tracing mechanism at predefined points in the program, but the support is experimental and not well supported by the current version of DynamoRIO. We attempted to insert the calls to start and stop tracing into the GHC runtime, but it simply caused the program to crash. With all of the negative performance effects we observed with Haskell programs we decided it was best to pursue a different path for trace-based optimization, and to attack directly the problem of building appropriate traces for Haskell programs.

5.5 Conclusion

Our investigation of the characteristics of Haskell codes running under DynamoRIO revealed three primary insights. First, we saw that Haskell codes spend a lot of time in the indirect branch lookup routine and that this causes a big hit to performance. Second, we saw that the traces generated by DynamoRIO are a decent size and that a few traces account for a large amount of the execution time. Third, DynamoRIO's trace building heuristics miss a major source of inefficiency in the Haskell codes - the relatively heavy use of indirect branches (when compared to more traditional codes such as SPEC). These inefficiencies are likely to also appear in some other kinds of

programs, such as OO-heavy codes. However, they may be ameliorated in the OO case by substantial amounts of inlining. The results presented here show that we need to reduce the amount of time spent in the IBL for Haskell programs if we are to get a performance improvement.

Our investigation of the traces built by DynamoRIO found that they were causing the program to spend a lot of time in the indirect branch lookup routine. Additionally, many extraneous traces were created that were not executed very often, including many traces in the garbage collector. These extra traces add to the overhead of running the program through DynamoRIO. The root of the problem is that the heuristic used for building traces is not very good for most Haskell programs. DynamoRIO marks traces heads as the targets of backward branches. This heuristic makes sense for finding loops in imperative programs, but is not very suitable for the code shape of Haskell programs.

The promising results from this study are that we can find traces in Haskell programs and that it takes few traces to account for the majority of the time spent executing traces. If we can find a way to build good traces without incurring the runtime overhead then we may be able to find some real optimization opportunities.

The next chapter describes Htrace, an LLVM trace-based optimizer for Haskell. Unlike DynamoRIO which builds the traces at runtime, Htrace uses a separate training phase for finding, building, and optimizing traces. This design is intended to get the benefits of optimizing over the scope of program traces without the performance penalty of building traces at runtime.

Chapter 6

Static Trace-Based Optimization of Haskell with Profile Data

In Chapter 4 we saw that low-level Haskell differs from code coming from traditional programming languages. These differences are primarily caused by lazy evaluation, higher-order functions, and the separately managed Haskell call stack. The differences manifest in low-level code as a preponderance of indirect jumps. The different code shape of Haskell programs limits the effectiveness of traditional compiler optimizations because it limits the scope of optimizations to small single functions. We proposed using runtime trace-based optimization to increase the scope available to the compiler, but found that the overheads of building and maintaining the traces at runtime was too large. Further, we saw that the current state-of-the-art in runtime trace collection for imperative languages, exemplified by DynamoRIO, does not capture the traces that seem critical to optimization of Haskell programs. In this chapter we explore the idea of building and optimizing traces in a separate offline phase.

We want to gain the advantage of larger program scope without suffering from the runtime overhead we experienced with DynamoRIO. There were three main problems with running Haskell programs through DynamoRIO. First, the tracing heuristic

caused it to build traces that did not correspond to important paths through the Haskell program. Second, it would build traces whose indirect branch targets changed, which caused the program to spend a lot of time in the indirect branch lookup routine. Finally, it was building traces through parts of the GHC runtime, such as the garbage collector, that are not part of the inefficiencies we are trying to target in this work. To combat these problems we built Htrace, a new trace-based optimizer that knows about the structure of low-level Haskell code. Htrace finds traces in a separate profiling run and uses them to restructure the program offline to avoid the runtime overhead of a dynamic optimizer.

In this chapter we describe the design of Htrace, its implementation in LLVM, and the performance results.

6.1 Design

In designing Htrace, we chose to separate the process into three phases: finding traces in the program, restricting the low-level code around those traces, and optimizing the restructured program. This separation allows for experimentation with different strategies for each task without a tight coupling between them. After our experience with performance problems when trying to build and optimize the traces at runtime, we chose to use a separate profiling run to find frequently executed paths. Once the paths have been discovered we use a separate pass to build and optimize traces along these paths.

Using a separate training run to find program traces is a tradeoff of precision for speed. The traces will be fixed after the profiles are collected. If the traces take paths that are largely data dependent then the program will gain no benefits when run with different data and may perform worse because of misguided assumptions that are enshrined in the optimized code. However, this problem—the need for representative

training data—arises in all feedback-driven offline optimization. Building the traces at runtime gives precise behavior at the cost of diverting execution time to finding and maintaining the connections between these traces, and re-optimizing the code during every execution. As we saw with DynamoRIO, these overheads can be significant.

Figure 6.1 shows the overall structure of the Htrace design. The diagram starts from the low-level code in the bottom left corner, which is produced by GHC and LLVM. Note that some of the external libraries, in low-level code form, are also used as an input to Htrace. The design can be broken up into three main components: trace finder, trace builder, and trace optimizer. The trace finder is in the left hand side of the figure and is primarily composed of the “Trace Runtime” and “Trace Callbacks” components. After finding the traces, they are written in an external form to a trace profile file. The profile data is read by the “Trace Builder” to restructure the program around the traces. Finally the “Trace Optimizer” shown in the figure is used to optimize the restructured code.

In this thesis we focus on how to find and build the traces. The trace finder takes a program and instruments it to find the traces in a program. The program is run and the traces we find are recored in a separate file for later usage. Once the traces have been found we modify the original program by instantiating the traces and rewriting the code to jump to the trace entries. Once the traces are fully built, we optimize them using standard compiler optimizations. Each of these components are described in more detail below.

6.1.1 Finding Traces

To find traces in a program we must be able to monitor the program as it executes. We do this by inserting instrumentation into the program that will record the execution sequence and call back into our Trace Runtime as specific points. The instrumentation serves as the mechanism for inspecting the program execution. One important

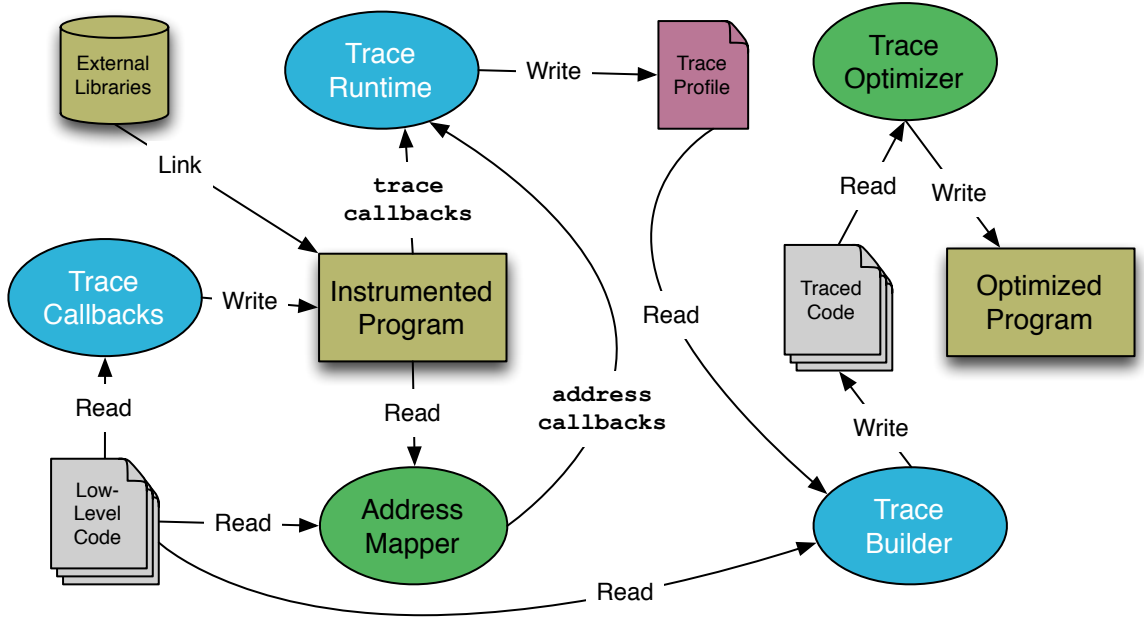


Figure 6.1: The design of the Htrace system. In this thesis we focus on how to find the traces (trace callbacks, trace runtime), and how to restructure the program around the traces (trace builder).

question to address is what program locations should serve as the potential starting points for program traces.

A *trace header* is a program point that can serve as the starting location for a trace. DynamoRIO uses the targets of backward branches as trace headers. The reason they chose this policy is to start the traces at loop headers so that the trace will encompass as much of the loop as possible and will be focused on the hot parts of the program. This definition does not work well for Haskell programs since the loops will be encoded as recursive functions. Because of lazy evaluation these loops may actually include portions of multiple functions. The notion of a backward branch does not make much sense in this situation because the different functions can be laid out in arbitrary locations in the code. We need a different policy for marking trace heads in Haskell.

The policy we adopted was to mark as trace headers the low-level functions that serve as the entry points for the high-level Haskell functions and thunks. This defini-

tion will exclude any low-level function that is generated because of lazy evaluation as an internal continuation point for a Haskell function. In addition, we exclude as trace headers any low-level functions from the runtime that are used to implement GHC's execution model. We want our traces to include these runtime functions, but we do not want our traces to start from them since they are typically called from many different call sites.

The trace header policy was chosen to have traces start at natural locations in the original Haskell program. Many different policies are possible and could drastically change the traces found by the profiler. Our policy lets the programmer's design and decomposition influence the choice of trace headers. Program loops will be represented by recursive functions, so the stated policy will have no trouble finding the loops. Further, the programmer decomposed the original problem into the units found in the source code. We rely on their original intuition to guide our choice of headers rather than marking headers in arbitrary code locations that are created to implement Haskell's execution model. Our choice seem to produce good traces on the sample programs, so we have not experimented with alternate policies. It would be interesting to explore different trace head policies, but that task is out of the scope of this thesis.

Trace Instrumentation

Now that we have identified the policy for choosing trace heads we can describe the algorithm used to insert instrumentation for collecting the traces. The main goal of the instrumentation is to record the sequence of basic blocks executed by the program. Conceptually we just need to insert a callback to the trace-building runtime at the top of each block to record its execution. The practical difficulties arise when we have direct function calls to code that we cannot see and indirect calls to any function.

Direct function calls cause problems when we cannot instrument the source code for the called function. This situation arises with any direct call to an external

function. These functions will cause a break in the sequence of blocks that make up the trace. Further, the called function could call back into another function that we did instrument which would make it look like the trace is continuous between the two functions even though it is separated by one or more intervening functions. At direct function calls to functions that we are unable to instrument we need to insert a callback to the tracing runtime to note that we have a break in the trace at the current location.

Indirect function calls cause a similar problem as direct function calls to external functions. The difference with indirect function calls is that we do not know the called function when we are inserting the instrumentation so it is not clear if the call will cause a break in the the trace. We need to check at runtime whether or not we have instrumented the target of the function call. If we have instrumented the target then no break in the trace is needed. Otherwise, we need to record a break in the trace just as with a direct function call.

Part of the difficulty in implementing the trace-breaking scheme for indirect function calls is finding a way to match function addresses used in an indirect call to the actual function being called. We describe an implementation in Section 6.2 that modifies the LLVM JIT to make the correct associations. For now, we will assume that we have the correct hooks in place for notifying the Trace Runtime about the association between a function address and its implementation.

The algorithm for inserting instrumentation is shown in Figure 6.2. There are three program constructs that need instrumentation: functions, basic blocks, and call sites.

Each function needs to be assigned a unique function number. This function number is used by the runtime to determine when function calls require a break in the trace. The function number can be added as an annotation in the function code so that it can be read by later passes. The function number annotation is used in our

```

1   for each Function F
2       add function number annotation
3
4   for each BasicBlock B
5       if IS_TRACE_HEAD(B)
6           insert call to tracer_trace_head(BasicBlockNumber(B))
7       else
8           insert call to tracer_trace_path(BasicBlockNumber(B))
9
10  for each CallSite C
11      if C.isIndirect?
12          add call to tracer_check_trace_break(Address(C.target))
13      elif C.target.isOnlyDeclaration?
14          add call to tracer_break_trace(FunctionNumber(C.target))

```

Figure 6.2: Algorithm for inserting instrumentation to build program traces.

implementation to map function addresses to implementations.

In addition to numbering functions in the program, we also need to number the basic blocks. The basic block numbering is communicated directly to the trace runtime through callback routines so it does not need to be recorded as a separate annotation in the code. All of the basic blocks in the program get a call back to the trace runtime at the top of the block. If the trace policy says that the block should be considered as a trace head then we insert a call back to the runtime to indicate we have a potential trace head. Otherwise, the block is not a header so we insert a callback to record that we are passing through the block. Both of the basic block callbacks pass the basic block number for the block containing the callback.

The last construct requiring instrumentation are the call sites in the program. At each call site we need to check two things. First, if the call site is indirect we have to let the trace runtime know that we are about to go through an indirect call and pass the address of the function that we are calling. If the call is direct then we need to see if we have an implementation of the function being called. When the called function is external to our program then we insert a callback to the runtime so that

it knows we are about to have a break in the current trace.

After instrumenting the program we are ready to run it and collect the program traces. The callbacks we inserted are used to communicate the execution path of the program to the trace runtime. Next we describe how the runtime uses the information in the callback routines to build traces.

Trace Runtime

The Trace Runtime receives callbacks from the program as it executes and uses the provided information to record program traces. The runtime is modeled after a simple state transition system. It maintains some internal state which dictates how it responds to the callbacks from the program. The runtime will transition between states based on its current state and the new data received from the callback. Figure 6.3 shows the state transition diagram for the trace runtime.

The runtime has three states as shown in the diagram: Profile, Trace, and Shadow. The Profile state is the starting state of the system and is used to collect data about which parts of the program are executing frequently. The Trace state is used when we are actually recording the execution path for a program trace. The Shadow state is used for subsequent entries into a trace that we have previously recorded. It can be used to collect additional profiling data about the trace or to re-record the trace with a new path. We currently only use the Shadow state to collect trace-exit profiling data.

Figure 6.4 shows the trace runtime callbacks used to record program traces. The `tracer_trace_head` callback is used to signal to the runtime that we are about to execute a block that has been marked as a potential trace header. If we are in the Trace or Shadow state then we can simply extend those traces with the current block. Otherwise, we must be profiling and so might need to transition to a new state. First, we increment the hotness counter for the block and check to see if it is considered hot

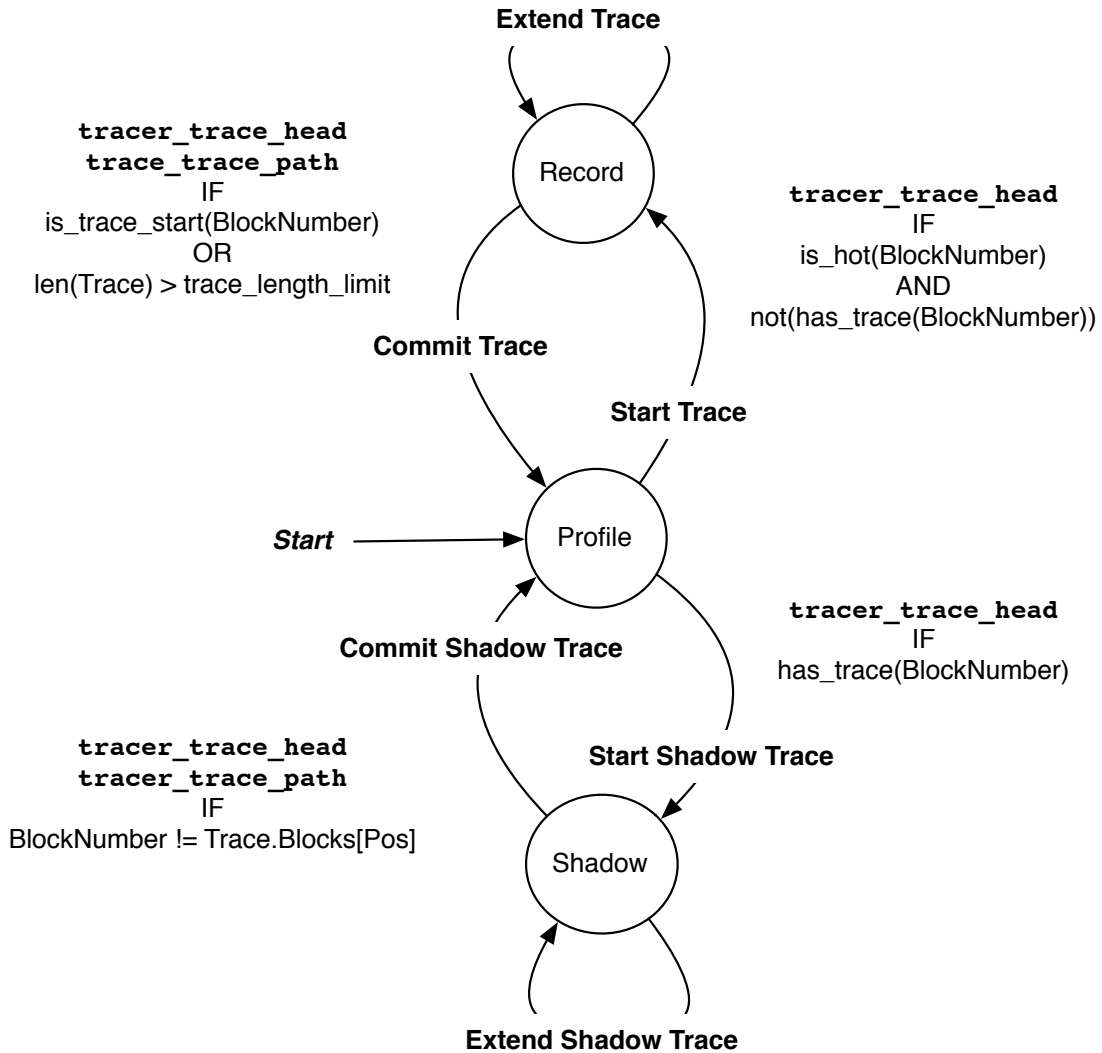


Figure 6.3: State transition diagram for trace runtime. The nodes in the diagram represent the states of the trace runtime. The edges are labeled with actions that are taken when moving between the states. Next to the edges the conditions for the transition are listed underneath the callbacks that can cause the transitions.

enough to start at trace. We also check to make sure that the block is not already part of a trace. We avoid starting a trace from a block that is already contained in another trace. Starting traces from multiple trace headers is problematic when these headers are all part of the same trace because we end up with multiple entries onto the trace. Without this restriction we would get one version of the trace for each trace header in a trace that loops back to its starting point.

If a header passes all criteria for starting a new trace we will start recording the trace from that block. Otherwise, we check to see if we have previously recorded a trace for this block. If we do find a previous trace we begin recording a shadow trace from this block.

The `tracer_trace_path` callback is used when entering a basic block that should not be used as a trace header. If we are in the Trace or Shadow state then we will extend the appropriate trace. Otherwise no action is necessary.

The `tracer_check_trace_break` and `tracer_break_trace` callbacks are used to signal potential and definite breaks in a trace. They are only used when in the Trace state and are otherwise ignored. The logic for handling the breaks is pushed into the `ExtendTrace` function which will be discussed shortly.

Finally, the `tracer_map_target_address` callback is to associate an address to a known function. We need this association to handle indirect calls so that we know whether we should break the trace on the call or not. The `ADDR_MAP` is used inside the `ExtendTrace` function when it is called with an address from the `tracer_check_trace_break` callback.

The `ExtendTrace` function is called when we are in the Trace state and we get a callback to notify the runtime that we entered a block or are about to (potentially) break the trace with a function call. The code for the `ExtendTrace` function is shown in Figure 6.5. The notation `{T|Payload}` is used to indicate a trace record that is tagged with the tag T and has a payload containing the Payload data.


```

1  tracer_trace_head(BB)
2      case State of
3          Trace ->
4              ExtendTrace(BB)
5          Shadow ->
6              ExtendShadowTrace(BB)
7          Profile ->
8              HOTNESS(BB)++
9              if HOT(BB) && NOT(IS_PART_OF_TRACE(BB))
10                 StartTrace(BB)
11                 elif HAS_TRACE(BB)
12                     StartShadowTrace(BB)
13
14  tracer_trace_path(BB)
15      case State of
16          Trace -> ExtendTrace(BB)
17          Shadow -> ExtendShadowTrace(BB)
18
19  tracer_check_trace_break(Addr)
20      if State == Trace
21          ExtendTrace(Addr)
22
23  tracer_break_trace(FN)
24      if State == Trace
25          ExtendTrace(FN)
26
27  tracer_map_target_address(FN, Addr)
28      ADDR_MAP(Addr) = FN

```

Figure 6.4: Trace runtime callbacks.

```

1   StartTrace(Header)
2       State = Trace
3       initialize new Trace structure
4       append {B|Header} to Trace
5
6   ExtendTrace(With)
7       case With of
8           BasicBlock(BN) ->
9               append {B|BN} to Trace
10              if IS_HEADER(BN)
11                  IS_PART_OF_TRACE(BN) = True
12              if IS_TOO_LONG(Trace) || IS_LOOP(BB, Trace)
13                  CommitTrace()
14           Function(FN) ->
15               append {F|FN} to Trace
16           Address(Addr) ->
17               append {A|Addr} to Trace

```

Figure 6.5: The `ExtendTrace` routine. The notation `{T|Payload}` is used to indicate a trace record that is tagged with the tag `T` and has a payload containing the `Payload` data

To start a new trace we set our current state to `Trace` and initialize a new trace structure. We then record the header as the first block in the trace tagged with the tag `B` so we know it is a block number. Each time `extend trace` is called, we check the type of the data being used to extend the trace. If we are given a basic block number, then we add a new trace record with that block number. If that block is also a header we have to remember that it is now part of a trace so that we do not start a new trace from it later. Finally, we check the termination conditions. When the trace becomes too long or it becomes a loop we stop recording the trace and commit it to disk. The `ExtendTrace` routine could also be called with a function number to indicate a break in the trace or an address to indicate a potential break. Either way we append the data to the trace by inserting a trace record with an appropriate tag. The record will be processed once the trace is committed.

Committing a trace means processing the trace records to ensure we have an unbroken sequence of blocks and converting the data to a format suitable for external

```

1   CommitTrace()
2       State = Profile
3       for Record(R) in Trace
4           case R of
5               {B|BB}  -> save {B|BB}
6               {A|Addr} -> if Addr not in ADDR_MAP then save {G|Addr}
7               {F|FN}  -> save {G|FN}

```

Figure 6.6: The `CommitTrace` routine.

storage. Figure 6.6 shows the `CommitTrace` routine. First we set the state to `Profile` because we are done recording the trace. Next we look at each record we recorded while building the trace. The records are consolidated so that there are only two allowable tags. The `B` tag is for basic blocks in the trace and they are recorded with the basic block number as the payload. The `G` is for a gap in the trace to indicate that the trace is broken. The consolidation is straightforward except for the records containing addresses for indirect function targets. We must lookup the address of the function in our `ADDR_MAP` to see if we know what function corresponds to that address. If the function is known, then there is no break in the trace and the next `B` record can be safely added. If the address is unknown then we must insert a trace gap record since the next block record comes after a deviation into unknown code.

Once a trace has been recorded it could potentially be entered many times in the future. Shadow traces are a way to monitor the subsequent executions of a trace to either collect statistics or modify the existing trace. The primary routine for handling a shadow trace is the `ExtendShadowTrace` function shown in Figure 6.7.

To start a new shadow trace we set our state to `Shadow` and then find the existing trace that starts at the given header block. We assume that each trace has a counter to keep track of how many times we have entered the trace, and we increment that counter when we start a new shadow trace. We keep track of the current position in the trace using the `ShadowPosition` counter which we initialize to one (assuming

```

1   StartShadowTrace(Header)
2       State = Shadow
3       ShadowTarget = existing trace structure starting at Header block
4       ShadowTarget.Entries++
5       ShadowPosition = 1
6
7   ExtendShadowTrace(BB)
8       if ShadowTarget.Records[ShadowPosition] != {B|BB}
9           ShadowTarget.Exits[ShadowPosition - 1]++
10          CommitShadowTrace()
11      elif ShadowPosition is last trace record of ShadowTarget
12          ShadowTarget.Exits[ShadowPosition]++
13          CommitShadowTrace()
14      ShadowPosition++
15
16  CommitShadowTrace()
17      State = Profile

```

Figure 6.7: The shadow tracing routines.

zero-based indexing of trace records).

Each time we encounter a new block on the shadow trace the `ExtendShadowTrace` function is called. At that point, we check to see if the current block matches the block stored in the trace we are shadowing. If the block does not match, we have found an early exit for the trace which means that the previous block in the trace did not flow to the current block. We increment the exit counter for the previous block to indicate that we took an early exit from the trace at that block. If we have reached the final block of the trace then we increment the exit counter for that block to indicate we successfully made it through the entire trace. Once we have found an exit point of the trace (early or otherwise) we will go back to the Profile state.

When the program finishes executing we can write out all the traces we have found in a suitable external format. The trace file can then be read to report on statistics of the traces we found and to actually modify the code to take advantage of these traces. The next section describes how we use the trace records and rewrite the original program to take advantage of these program traces.

```

1   for each trace starting at function root
2       # Clone trace functions every time it occurs on the trace
3       i = 0
4       for each occurrence of function f in trace
5           f_trace_i = clone f
6           i += 1
7
8       # Replace calls on trace to stay on trace
9       i = 0
10      active_function = f_trace_i
11      for each block in trace
12          if block contains call to function f
13              find corresponding block in active_function
14              if call is indirect
15                  insert check to make sure target is still f
16              replace call to f with call to f_trace_i
17              active_function = f_trace_i
18              i += 1
19
20      # Inline cloned functions into caller
21      for i from 1 to size of trace
22          inline f_trace_i into caller
23
24      # Replace calls of root function
25      for each use of function root
26          replace with use of f_trace_0
27

```

Figure 6.8: Algorithm for instantiating traces.

6.1.2 Building Traces

After finding the traces in a program we must actually modify the code to take advantage of this knowledge. In this section we describe a simple algorithm that uses the profiling data from the previous section to instantiate the traces. The basic strategy for building traces is to clone all of the functions on the trace and then inline the cloned functions into their call site on the traces. Figure 6.8 shows the algorithm for building traces.

A trace consists of a sequence of basic blocks that come from one or more functions. The traces found by the profiler are instantiated one at a time. For each trace we

first clone every function on the trace every time it occurs on the trace. For example, if a function appears twice on the trace we will produce two distinct clones of the function. Next we need to modify the calls that reside in the trace blocks so that they target our newly cloned functions.

We process each of the original blocks on the trace and look for the blocks that contain calls. When the trace block has a call we find the corresponding block in the function we cloned for the trace. If the call is an indirect call we need to insert a check to make sure that the target matches the target that we found when recording the trace. The check compares the target of the call to the address of the original function that was the target when trace was found. If the address is a match then we insert a direct call to the cloned function. If the target has changed since we recorded the trace then we must execute the indirect call to ensure we jump to the correct target. If the call is a direct call then we can simply replace the call to point to the cloned trace function. Either way update the `active_function` to the targeted trace function so that we know where to search for corresponding block for the next call.

Once all of the call instructions have been modified to keep execution on the trace we can inline the trace functions. Each of the cloned functions will only be called from one site (e.g. the trace we are building) so we can easily inline the function and then delete the cloned copy. We know that the cloned function will only be called from one site on the trace because we have cloned each function each time it occurs on the trace. We need to keep the original functions around since they may still be called from other locations.

After inlining the trace functions we can replace uses of the root of the trace with the newly cloned root function. This update will point all users of the original function to the trace. Since we only allow one trace per header this policy is reasonable, but could lead to excessive trace exits if the callers do not follow the trace path. We could use a more fine-grained policy for replacing uses of the trace root, but have not

explored that area in this thesis.

We currently instantiate all traces that we found during profiling. However, there are a wide range of policies that could be used for deciding which traces to instantiate. The shadow tracing technique described above can be used to collect additional profiling data such as trace entry and completion rates. We also currently instantiate both loop and non-loop traces, but this is another choice that could be changed. There is much room in the future for exploring different trace building policies. We have described a basic algorithm that is easy to implement and produced good results for several benchmarks and generally avoids degenerate behavior.

6.1.3 Optimizing Traces

Once the traces have been found and instantiated it is time to optimize them. In this thesis we rely on the optimizations implemented by LLVM to improve the performance of our traces. Our main contention is that Haskell programs will benefit from running traditional optimizations over the increased scope of a program trace. After building the traces as described above, the trace will be contained in a single function. This function can be optimized using all of the traditional compiler optimizations implemented by LLVM.

Although we fully rely on existing LLVM transformations to improve performance, an easy trace-based transformation to implement would be to layout the basic blocks in the function according to their position in the trace. We already have this information available when building the trace so it would be conceptually simple to layout the blocks so that all of the trace blocks are grouped together at the entry of the function in the order they appear on the trace.

Our strategy of reusing the LLVM optimizations for traces has advantages and disadvantages. The primary benefit is that we do not have to re-implement any of these optimizations and we have access to a wide variety of transformations. The

major disadvantage is that we do not run any trace-specific optimizations. We have effectively increased the scope of optimization, but have not provided any guidance to the compiler about the likely path through the function. If LLVM had implemented any profile-guided optimizations we could seamlessly take advantage of those optimizations using the profiling data we collected when finding the traces.

6.2 Implementation

In this section we describe how we implemented the Htrace design described above. The implementation can be broken down into three main parts: changes to GHC, changes to LLVM, and a way to bind it all together. First, we describe some modifications to GHC. The modifications were primarily needed in the build system so that we could get access to the low-level code. The LLVM changes revolve around adding the trace instrumentation and building the traces according to the algorithms given in Section 6.1. Finally we describe how to combine all of the pieces to get a working system.

6.2.1 GHC Modifications

The modifications to GHC are mostly to the build system and not to the actual code for the compiler. The changes we needed to make in the build system were already supported as options. We are documenting the changes here to make it clear what options we use to build GHC. We also had to make some small changes in the declarations of some C functions in the GHC runtime to enable them to be dynamically linked into the LLVM interpreter. The necessary changes are described in detail below.


```

1 SRC_HC_OPTS          = -H64m -O
2 GhcStage1HcOpts     = -O -fasm
3 GhcStage2HcOpts     = -O2 -fllvm -keep-llvm-files
4 GhcLibHcOpts        = -O2 -fllvm -keep-llvm-files
5 GhcLibWays = v dyn
6 GhcEnableTablesNextToCode=NO
7 INTEGER_LIBRARY     = integer-simple

```

Figure 6.9: GHC `build.mk` file used to control the build options needed for trace experiments.

Generating and Keeping LLVM Files

We need to generate LLVM files for all of the libraries used by GHC as well as any of the hand-written CMM files used by the runtime. Figure 6.9 shows the `build.mk` file we used that controls the build options for GHC. The `GhcStage2HcOpts` and `GhcLibHcOpts` variables control the options used when building the final version of GHC. We passed the `-O2` option when compiling the Haskell files to ensure that we run the standard set of high-level optimizations GHC has available. The `-fllvm` and `-keep-llvm-files` flags are used to generate the LLVM IR for the Haskell source files.

We also need to generate the LLVM IR files for the hand-coded and auto-generated CMM files written for the GHC runtime. The runtime files we include in program traces are listed in Figure 6.10. These files include operations such as updating a thunk with an indirection after it has been evaluated as well as the code needed for applying unknown functions. To generate the LLVM files for the runtime CMM files we must modify the `ghc.mk` file in the runtime subdirectory of the GHC source code. The necessary changes are listed in Figure 6.11. It is important that GHC itself is built using the CMM code as compiled by LLVM. Due to a performance regression in the LLVM backend, the native code generator currently does a better job of compiling the CMM files. If the LLVM backend is not used when compiling the CMM files then

```
1 Apply.cmm
2 AutoApply.cmm
3 Exception.cmm
4 HeapStackCheck.cmm
5 PrimOpts.cmm
6 StgMiscClosures
7 StgStartup.cmm
8 StgStdThunks.cmm
9 Updates.cmm
```

Figure 6.10: GHC rts CMM files used in building traces.

```
1 rts/Apply_HC_OPTS += -fllvm -keep-llvm-files
2 rts/dist/build/AutoApply_HC_OPTS += -fllvm -keep-llvm-files
3 rts/Exception_HC_OPTS += -fllvm -keep-llvm-files
4 rts/HeapStackCheck_HC_OPTS += -fllvm -keep-llvm-files
5 rts/PrimOps_HC_OPTS += -fllvm -keep-llvm-files
6 rts/StgMiscClosures_HC_OPTS += -fllvm -keep-llvm-files
7 rts/StgStartup_HC_OPTS += -fllvm -keep-llvm-files
8 rts/StgStdThunks_HC_OPTS += -fllvm -keep-llvm-files
9 rts/Updates_HC_OPTS += -fllvm -keep-llvm-files
```

Figure 6.11: GHC rts/ghc.mk file used to generate the LLVM IR for the CMM files used by the GHC runtime.

the programs compiled by GHC have an unfair advantage from this known issue.

Using the integer-simple Library

The Haskell `Integer` type represents arbitrary precision integers, as opposed to the machine sized integers of the `Int` type. GHC implements the `Integer` type using GMP, which is the GNU Multiple Precision Arithmetic Library [GMP \[2011\]](#). The GMP library provides a fast implementation of multi-precision arithmetic, but it is an external library that requires a tight integration with GHC's allocator. We chose to use an alternate library for integer arithmetic that is provided by GHC. The `integer-simple` library is written in pure Haskell and provides an alternative to GMP. Because it is written in Haskell we can more easily gain access to the LLVM

IR using the same mechanisms described above for the other library code.

The disadvantage of the `integer-simple` library is that it is much slower than GMP, particularly when the integers grow large. We could have used the GMP library, but it would have required more effort to gain access to all of the LLVM IR needed for integer operations. Using the simple library allowed us to quickly have access to all the LLVM code used for integer operations.

One advantage of our overall approach to building traces using LLVM IR is that any code for which we have LLVM IR can be integrated into the program traces. Our design opens the door to building and optimizing traces across multiple languages. In this thesis we keep it simple and use the easily accessible integer implementation. We can use the simple library by setting the `INTEGER_LIBRARY` variable as shown in Figure 6.9.

Disabling Tables Next To Code

Tables next to code is an optimized closure layout used by GHC to reduce the overhead of jumping to a closure to evaluate it. As described in Chapter 3, GHC represents a closure by a pointer to the code to evaluate it along with the free variables needed by the evaluation. In addition to the evaluation code there are several more data fields that are stored for each closure. For example, the layout of the closure is needed by the garbage collector to distinguish the pointers from the non-pointers stored in the closure. These extra data fields can be shared by all closures of the same type. The combination of the evaluation code and extra data fields is called an *info table*.

Closures in GHC actually contain a pointer to the info table rather than just a pointer to the evaluation code. To optimize for the common case where we need to evaluate a closure, GHC uses a layout called tables next to code (TNTC) that places the data fields for the closure type directly before the code to evaluate the closure. The info pointer store in the closure can then point directly to the evaluation code

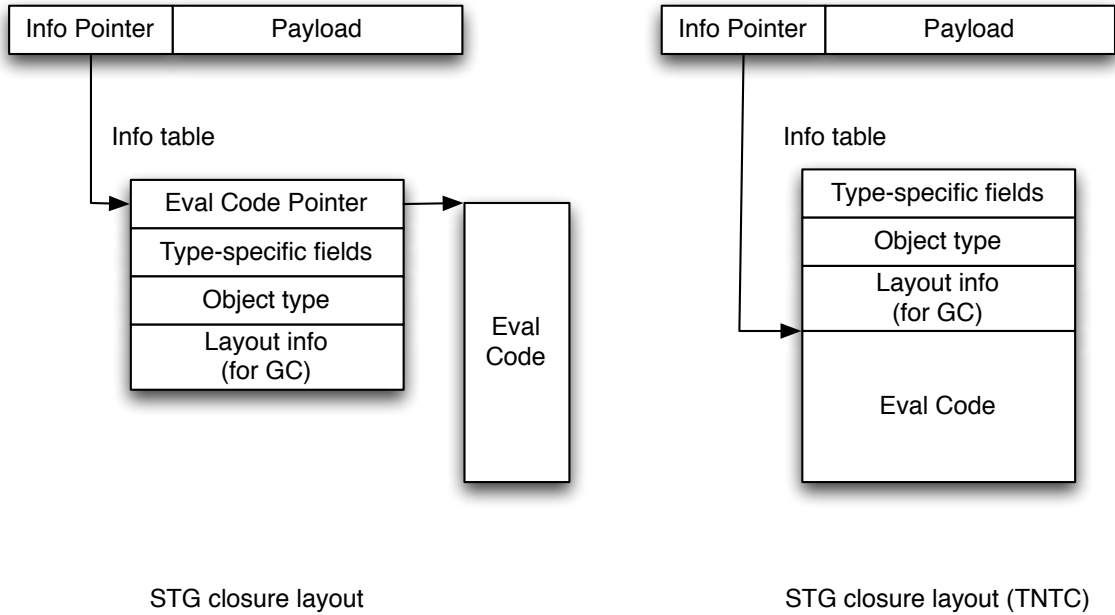


Figure 6.12: The tables next to code (TNTC) layout. The figure shows the difference between the non-TNTC layout (left) and the TNTC layout (right). We have disabled TNTC in this thesis because its implementation in the LLVM backend prohibits the merging of multiple LLVM IR files.

and the garbage collector can access the fields that it needs by using negative offsets from the info table pointer.

Figure 6.12 shows the difference between a non-TNTC and a TNTC layout of closures. In the non-TNTC layout the closure stores a pointer to the info table which then has another pointer to the evaluation code. The TNTC version stores a pointer directly to the evaluation code and places the other data fields directly before the code. The TNTC layout allows closure evaluation with a single indirection, compared to the double indirection need for the non-TNTC version.

Unfortunately, we cannot use TNTC for our trace implementation. The LLVM compiler does not directly support the layout of data next to code. The LLVM backend of the GHC compiler places the data and code in specific named sections and relies on the platform linker to order the sections correctly. These named sections conflict when we try to merge multiple LLVM IR files generated by the LLVM backend. While we could modify the backend to work across multiple IR files, the easy

and direct solution is to disable TNTC. Figure 6.9 shows how we disable TNTC by setting the `GhcEnableTablesNextToCode` variable.

Exposing More RTS Functions to the Dynamic Linker

A final change we had to make to GHC involved minor modifications to the source code for the runtime. The code for the runtime uses a GCC compiler extension that allows functions to be declared with `hidden` visibility, which means that they will not be public symbols in the final library. The Htrace system runs a Haskell program through the LLVM interpreter which needs to dynamically link with functions from the GHC runtime shared library. Several of these functions were declared with hidden visibility which was causing the dynamic linker not to find these functions in the shared library. We had to remove the hidden attribute from these function to enable the dynamic linker to find them when interpreting the Haskell program.

6.2.2 LLVM Modifications

We used the LLVM compiler framework to implement the Htrace system design described in Section 6.1. The implementation diagram is shown in Figure 6.13. We implemented the three components shaded with the Htrace color, and slightly modified the `lli` program. `lli` is an interpreter that can directly execute LLVM bitcode files. Bitcode is the external representation of the LLVM IR, which is a low-level (e.g. near-machine level) code in SSA form. Section 6.2.3 describes how we get access to all the bitcode files through which we want to build traces. For now we assume that we have access to all the necessary files.

We start by running the `llvm-link` tool to combine all the bitcode files into a single module. The linked bitcode is read by the trace instrumentation pass and instrumented to insert callbacks to the trace runtime. We then use the `lli` tool to execute the linked bitcode. The bitcode will be JIT compiled by `lli` to speed up the

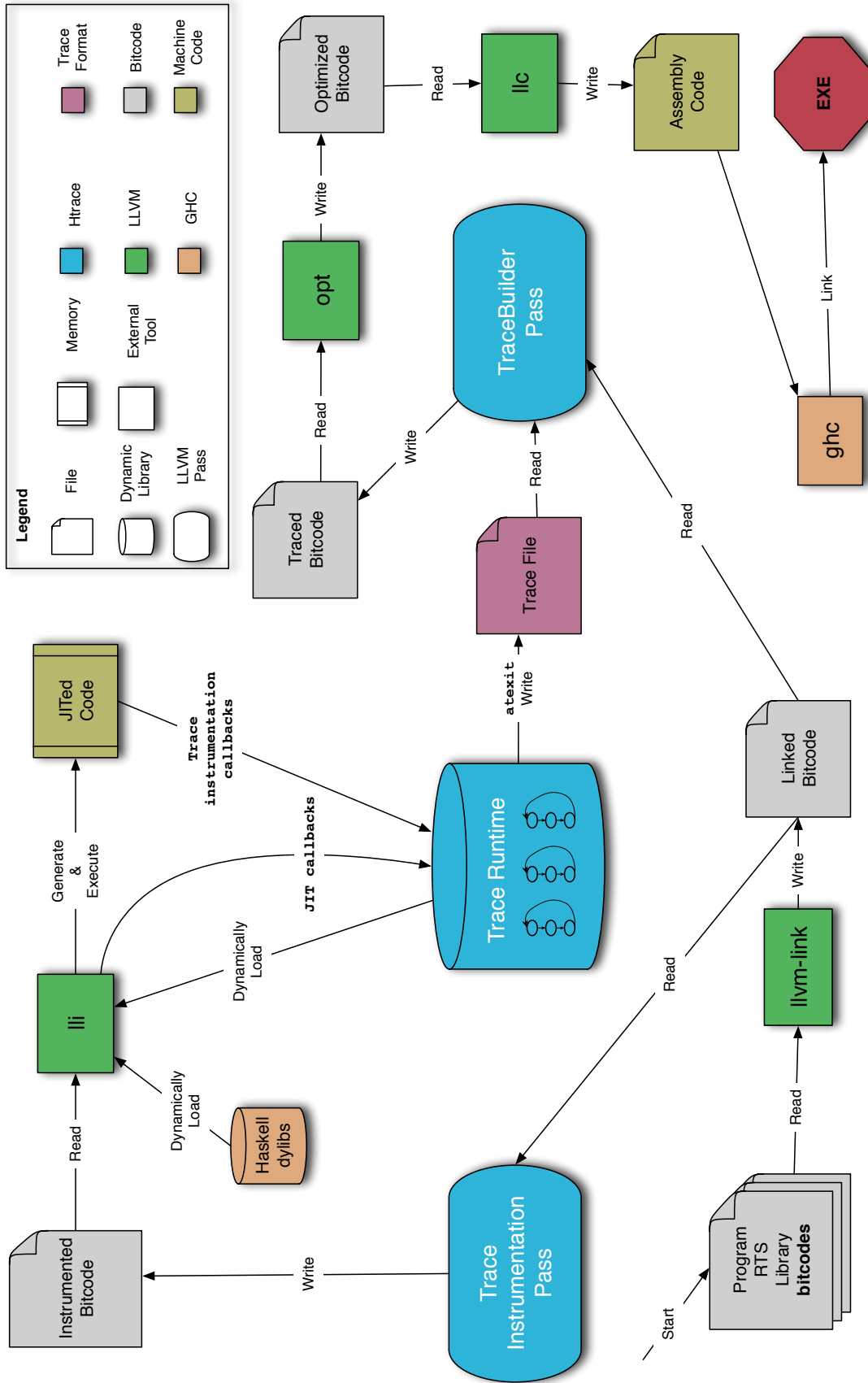


Figure 6.13: Htrace LLVM implementation.

execution time by translating the bitcode into machine code the first time a function is called. Any calls to functions not found in the bitcode will cause the JIT to look for the function with the dynamic linker. When those functions are called they will cause the program to execute from the appropriate library at full speed. As the program executes the JITed bitcode, it will callback to the trace runtime with the function calls inserted by the trace instrumentation pass. When the program finishes executing, the trace runtime writes out the traces to an external file.

Once the traces have been found and written to disk, we use another LLVM pass that modifies the original linked bitcode by building the traces found in the training run. The modified bitcode is then sent through the `opt` and `llc` tools to optimize the code. The `opt` program is LLVM's static optimizer and `llc` is the native backend that translates the optimized bitcode into assembly code. The `llc` tool writes out the final assembly code for the program. The assembly code is then fed into GHC, which uses the system assembler to produce object code, which is then linked with the appropriate Haskell libraries.

The primary modifications made to LLVM are the two new passes for inserting trace instrumentation and building traces, and a change to the `lli` tool for adding callbacks to the trace runtime. The Trace Runtime is implemented as a dynamic library that is loaded by the `lli` tool. Each of these components is described in more detail below.

Trace Instrumentation Pass

The trace instrumentation pass is implemented as a `ModulePass` in the LLVM compiler. The algorithm for inserting instrumentation closely follows the design given in Figure 6.2 on page 92. We begin by building a map that maps each function and basic block to a unique number. It is important that the numbering computed here be repeatable so that when we process the module again we will get the same num-

bering. To achieve a consistent numbering, we rely on LLVM to produce a consistent iteration order of all the functions in a module, and all the basic blocks in a function. A consistent numbering can then be achieved by incrementing counters for the function and basic-block numbers as we walk over the bitcode of the module. As long as we use the same bitcode for both instrumentation and trace building, we will have a consistent numbering. The numbering we compute for instrumentation is used by the Trace Runtime callbacks to identify the basic blocks as they execute. To rebuild the traces later we must map the block numbering back to the corresponding basic block.

Once the numbering has been computed, we can insert the annotations and callbacks needed by the trace runtime. For each function in the program we insert an `llvm.annotation` that records the function number. This annotation will be read by the `lli` JIT and used to notify the Trace Runtime about the mapping between the JITed function and its function number.

Next, we add instrumentation to each basic block. For each block we determine if it should be a trace header and if so we insert a call to the `llvm_tracer_trace_head` function passing the basic block number as a parameter. If the block is not a header then we insert a call to the `llvm_tracer_trace_path` function, once again passing the block number of the current basic block.

As described in Section 6.1.1, we have considerable latitude on what to mark as a potential trace header. We use a simple strategy based on naming conventions to decide which blocks are trace headers. GHC suffixes all function entry points with `_entry` and suffixes all continuation points with the string `_ret`. We mark as trace headers any function named with an `_entry` suffix. However, we exclude all functions beginning with `stg_`, since these are GHC runtime functions. We want to trace through the runtime functions, but not start traces with a runtime function because they can be called from many places.

We instrument function calls differently depending on whether they are direct or indirect calls. A direct call needs instrumentation only if it invokes a function outside the current bitcode module. We check this condition using the `F->isDeclaration()` predicate to see if the function is only a declaration with no body. If it is only a declaration then we will insert a `llvm_tracer_trace_break` callback before the function call.

We make one exception to the rule for instrumenting direct function calls. If the function is an intrinsic function, such as `llvm.sqrt`, then we will not insert a trace-break callback. Although we do not have the function bodies of the intrinsics we do not need to break the trace on these functions because they will not corrupt the control flow of the trace. We assume that the intrinsic functions do not re-enter the program except to return to the call point.

All indirect calls in the program need a callback to check that the call invokes a known function. For each indirect call in the program we insert a `llvm_tracer_check_trace_break` callback. We pass the address of the called function as the only parameter to the callback.

One final callback function is needed to complete the instrumentation pass. We insert a callback to initialize the trace runtime. The `llvm_start_trace_profiling_runtime` callback is added to the `main` function of the program. This callback allows the trace runtime to initialize itself before any code from the program is executed.

All of the callback functions described in this section are listed in Table 6.1. The trace instrumentation pass inserts all of the necessary callbacks except for the address mapping callback needed to handle indirect function calls. The `llvm_add_target_address` callback is described in the next section.

lli

The `lli` tool is the interpreter for bitcode files. We made a slight modification to it so that we could notify the trace runtime about the association between function addresses and function numbers. The runtime needs this information when deciding whether or not to break a trace at an indirect function call. The majority of the implementation of `lli` is contained in the `ExecutionEngine` library. Our changes were primarily made in the library, but we also added a flag to the `lli` tool to enable our callback changes.

In order to decide when to break a trace on an indirect call, we needed to build a mapping from function addresses to function numbers. Fortunately, the LLVM JIT already provides an event system that lets us listen for certain events and take actions when they occur. Our first attempt listened for the `NotifyFunctionEmitted` emitted event which is sent when the JIT compiles the code for a function. Once the function was emitted, we could see both the function body and its address. Unfortunately, the address we see at this event is an address in the code cache which turns out to not be the address we need.

When first loading a bitcode module, the JIT will emit a stub for each function in the module. The stub address is used by other functions that need to call the function. When the stub is first called, the function will be compiled and emitted to the code cache. The `NotifyFunctionEmitted` event sends the code cache address, but we actually need the stub address since that is the address used for all of the indirect function calls we are monitoring.

We added a new event called `NotifyResolvedLazyStub` and send the event when the function is compiled for a stub address. We have a custom listener that handles the event. When the event is received we have access to the bitcode for the function and the stub address that was just resolved. We read the function number annotation from the bitcode and then call the `llvm_add_target_address` callback passing the

Callback	Purpose	Location
<code>llvm_start_trace_profiling_runtime</code>	Initialize runtime	<code>main</code> function
<code>llvm_tracer_trace_head</code>	Trace header	Select functions
<code>llvm_tracer_trace_path</code>	Trace basic block	Every basic block
<code>llvm_tracer_check_trace_break</code>	Check indirect calls	All indirect calls
<code>llvm_tracer_trace_break</code>	Explicit trace break	All external calls
<code>llvm_add_target_address</code>	Update address map	<code>lli</code> JIT

Table 6.1: List of callback routines for the trace runtime

function number and stub address. The Trace Runtime will maintain the mapping and use it for deciding when to break the traces.

Trace Runtime

The Trace Runtime is a dynamic library that gets loaded by `lli` and provides an implementation for all of the callbacks listed in Table 6.1. The implementation of the callbacks closely follow the algorithm given in Section 6.1.1.

The runtime starts when the `llvm_start_trace_profiling_runtime` function is called. The Trace Runtime creates a new `Tracer` object that is used to encapsulate the tracing state and handle all the callbacks. Each time a callback function is executed, a corresponding method is called on the `Tracer` object. The `Tracer` object implements the state transitions shown in Figure 6.3. The reason for using an object to maintain state and record traces is so we can support concurrent callbacks from multiple threads. Each thread can maintain its own `Tracer` object in thread local storage. Although we currently only have single-threaded benchmarks, the implementation is flexible enough to support multi-threaded applications. In addition to initializing the tracer object, the startup function also registers a callback with the `atexit` function to dump the traces to a file.

Traces are stored in memory as a vector of `Trace` objects. Each trace contains a sequence of basic blocks along with any breaks in the trace. The traces are added to the vector when we take the `CommitTrace` transition out of the `Record` state. When

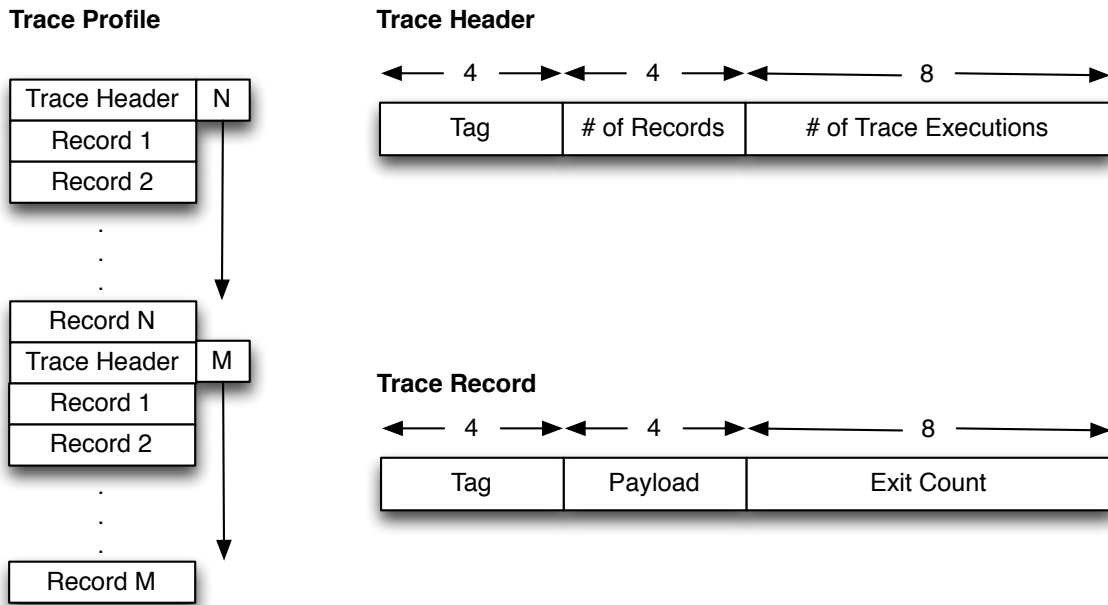


Figure 6.14: External format of trace records. Each trace has a trace header followed by a number of records as indicated in the header. The details of the trace header and trace records are shown in the right side of the figure. The numbers above the fields are the size in bytes.

entering the **Shadow** state, we lookup the corresponding trace in our vector of **Trace** objects. When the program completes the exit handler that we installed at startup is called and we use it to dump the traces to disk.

The traces are stored to disk in a compact binary format. The external trace format is shown in Figure 6.14. Each trace is stored with a header that indicates the number of records in the trace and the number of times that trace was entered during profiling. The trace records are tagged by the type of the record and contain a payload of data along with an exit counter for the number of times the trace was exited at this point. There are essentially two different types of trace records: breaks and blocks.

Figure 6.15 shows the C definitions for the trace record types. The **TraceGapRecord** is used to indicate a break in the trace. Its payload will be a function number if it was broken by a direct function call. The **TraceBlockRecord** and **TraceHeaderBlockRecord**

```

1   typedef uint64_t BigCounter;
2   struct TraceProfileHeader {
3       int TraceSize;
4       BigCounter NumHits;
5   };
6
7   struct TraceProfileRecord {
8       TraceProfileRecordType Tag;
9       union {
10          BasicBlockNumber BlockNumber;
11          FunctionNumber   FunctionNumber;
12      };
13       BigCounter ExitCount;
14   };
15
16   enum TraceProfileRecordType {
17       TraceGapRecord,
18       TraceBlockRecord,
19       TraceHeaderBlockRecord
20   };

```

Figure 6.15: C type definitions for external trace format.

tags are used to record the blocks along the trace. Their payload is a basic block number. We tag the records differently depending on whether the block was a header block or not, but currently do not make use of this information.

After the traces have been dumped to an external file, we can read them later in the LLVM pass that uses the profile data to build an increased optimization scope.

Trace Builder Pass

The final modification we made to LLVM was the inclusion of a new pass to restructure the program based on the trace profiling data. We follow the algorithm given in Figure 6.8. The most onerous part is maintaining the mapping between the blocks in the cloned trace functions and the blocks in the original trace. Maintaining the mapping requires some bookkeeping, but is otherwise manageable.

The first step of building traces is to find all of the functions that we need to clone.

We can do this by walking the trace and noting the places where two consecutive blocks come from different functions. Each time we see such a transition, we must clone the target function.

LLVM has direct support for cloning functions which makes it very easy to implement the cloning. After we clone a function we build a mapping between the blocks in the old function and the new function. This mapping allows us to get a handle on the basic block in the cloned function that corresponds to the basic block from the original function. Each time we have a transition between functions on the trace, we replace the call instruction in the cloned function so that it calls the newly cloned function that contains the next block on the trace.

To get all of the trace functions inlined into a single function in LLVM we mark the cloned functions with the `internal` linkage and the `alwaysinline` attribute. The LLVM inliner will always inline functions with the `alwaysinline` attribute. The `internal` linkage allows LLVM to delete the function once it has been inlined into all of its call sites.

After all the calls have been cloned for the trace we replace uses of the trace header function with the cloned version. The replacement is straightforward given the def-use chains automatically maintained by LLVM.

After the traces have been instantiated we optimize them using the standard LLVM optimizations in the `opt` and `llc` tools. The next section describes some of the details in how we put all of the different parts together to make the Htrace optimizer.

6.2.3 Putting It All Together

In this section we describe how we bring together all of the components to go from a Haskell program to an executable that has been optimized by Htrace. The procedure can be largely automated by scripting the interaction between all of the tools. There

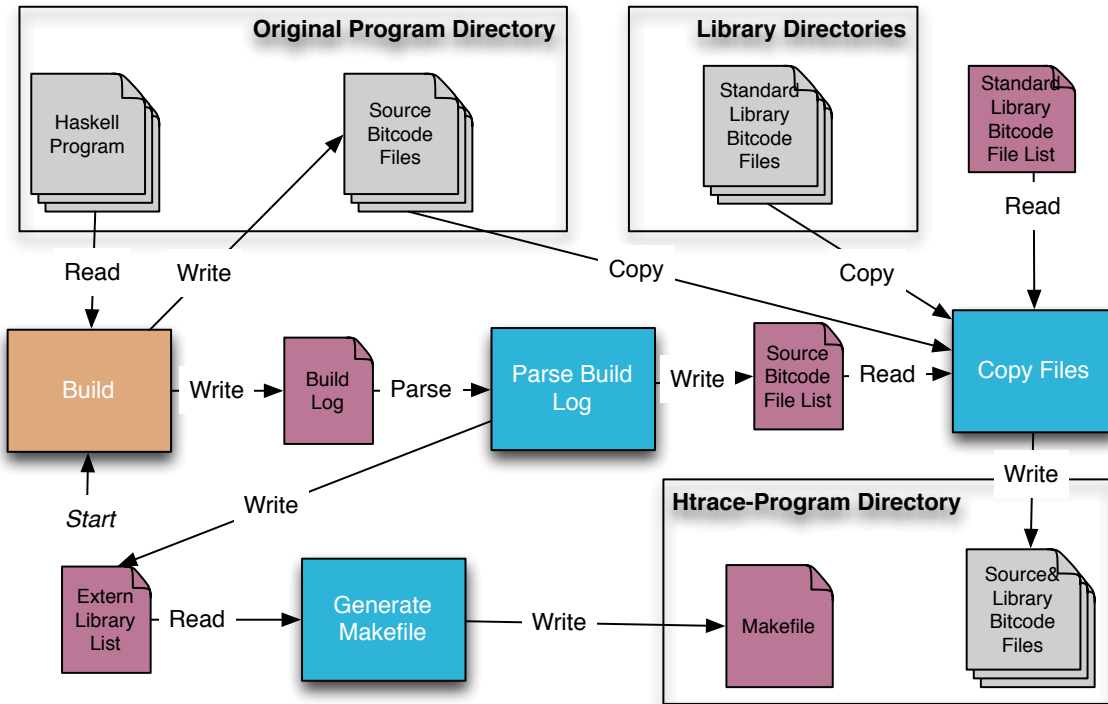


Figure 6.16: Preparing a Haskell program to use Htrace.

are four primary tasks we need to perform: gather the LLVM bitcode for the program source, gather the LLVM bitcode for the Haskell libraries, determine the external libraries used by the program, and generate a makefile to perform the build. The entire process is shown in Figure 6.16.

We start by assuming we have a Haskell program in its own directory. We build the program using `ghc -v --make -fllvm --keep-llvm-files` so that GHC will generate a LLVM bitcode file for each Haskell source file. The `-v` option is used to get verbose output from GHC which we save as a build log. The build log from the verbose output is parsed to get an explicit list of all the bitcode files generated and the external libraries that are linked into the program. These lists are recorded in temporary files that will be read by later tools.

We use a separate Htrace directory to contain all of the build artifacts. We copy all of the bitcode files that were generated from the source program to this new directory. Additionally, we need to copy bitcode files for libraries through which we want to be

able to collect traces. We can gather the required libraries by parsing the build log. In the case of the Fibon benchmarks, the source directories contain the source for all the libraries they use, except for some standard Haskell libraries. We keep a fixed list of the standard Haskell library files that are copied when initializing an Htrace program.

The list of standard library files is shown in Figure 6.17. We copy the entire implementation of the `prim`, `containers`, and `integer` libraries. From the `base` library we include the files that make up the Haskell Prelude, which is a module containing many useful functions that are automatically available to all Haskell programs.

The final step in Htrace preparation is the generation of a makefile. The makefile is a convenient way to orchestrate all of the various steps needed to go from a set of LLVM bitcode files to an executable that has been optimized to take advantage of the common program traces. Essentially, the makefile encodes the dependencies depicted in Figure 6.13 so that we can simply type `make` in the program's Htrace directory and have it run the correct sequence of commands that will profile, restructure and optimize the code. To correctly generate the makefile we need to know which external libraries are used by the program. These libraries are added to the dynamic linker's search path when running `lli` and also used in the final linking of the executable.

6.3 Results

In this section we present the data we collected to assess the effectiveness of Htrace. The results are presented in three areas. First, we examine the performance gain we achieve by restructuring the program to take advantage of the program traces. Next we look at properties of the traces found when using our trace heuristics. Finally, we discuss the effect of varying the hotness threshold that triggers a new trace.

Our results show that we can achieve a speedup of up to 86% over an optimized


```

1 packages = {
2   'base' :
3     ['libraries/base/GHC/Base.ll',
4      'libraries/base/Data/Tuple.ll',
5      'libraries/base/GHC/Show.ll',
6      'libraries/base/GHC/Enum.ll',
7      'libraries/base/Data/Maybe.ll',
8      'libraries/base/GHC/List.ll',
9      'libraries/base/GHC/Num.ll',
10     'libraries/base/GHC/Real.ll',
11     'libraries/base/GHC/ST.ll',
12     'libraries/base/GHC/Arr.ll',
13     'libraries/base/GHC/Float.ll'],
14
15   'prim' :
16     ['libraries/ghc-prim/GHC/Classes.ll',
17      'libraries/ghc-prim/GHC/CString.ll',
18      'libraries/ghc-prim/GHC/Debug.ll',
19      'libraries/ghc-prim/GHC/Generics.ll',
20      'libraries/ghc-prim/GHC/IntWord64.ll',
21      'libraries/ghc-prim/GHC/Magic.ll',
22      'libraries/ghc-prim/GHC/Tuple.ll',
23      'libraries/ghc-prim/GHC/Types.ll'],
24
25   'containers' :
26     ['libraries/containers/Data/Graph.ll',
27      'libraries/containers/Data/IntMap.ll',
28      'libraries/containers/Data/IntSet.ll',
29      'libraries/containers/Data/Map.ll',
30      'libraries/containers/Data/Sequence.ll',
31      'libraries/containers/Data/Set.ll',
32      'libraries/containers/Data/Tree.ll'],
33
34   'integer' :
35     ['libraries/integer-simple/GHC/Integer/Logarithms/Internals.ll',
36      'libraries/integer-simple/GHC/Integer/Logarithms.ll',
37      'libraries/integer-simple/GHC/Integer/Simple/Internals.ll',
38      'libraries/integer-simple/GHC/Integer/Type.ll',
39      'libraries/integer-simple/GHC/Integer.ll'],
40 }

```

Figure 6.17: Standard library files used for Htrace programs. The string on the top is the name of the Haskell library. The list below are the files that are copied for that library.

LLVM version. We get an average (geometric mean) speedup of 5% for the Fibon benchmarks. The hotness threshold for building traces can have a large impact on the quality of the traces found. In general, higher hotness thresholds are better but there is a significant amount of variation for the best threshold across all benchmarks.

The results show that Htrace can be very effective at improving performance when operating with the existing settings. The performance can likely be further improved by exploring different trace parameters and developing optimizations that specifically operate on traces.

6.3.1 Performance

In this section we discuss the performance impact of Htrace. The baseline for comparison is GHC using the LLVM backend. GHC has been modified as described in Section 6.2.1. We are measuring the performance impact that can be attributed to Htrace, so we compare against an equivalent GHC using the standard LLVM backend. The speedup measurements compare the CPU time used by the mutator. Recall that the mutator is the portion of the execution outside of the storage management code in the runtime. The CPU time provided a more consistent measurement than wall clock time on the MacBook Pro that was used to collect the results. The details of the machine are described in Table 2.2 on page 15. We measured the change in mutator time so that we could detect performance improvements even in GC-heavy benchmarks. Also, the low-level optimizations we are investigating do not typically change the time spent in the garbage collector because they do not change the amount of data allocated in the heap. The absolute performance improvement will depend on how much time is spent in the mutator.

We used the same data set for both the profiling run used to build traces and the performance run used to collect results. Using the same input for training and performance shows the limits of the performance improvement we can expect to achieve.

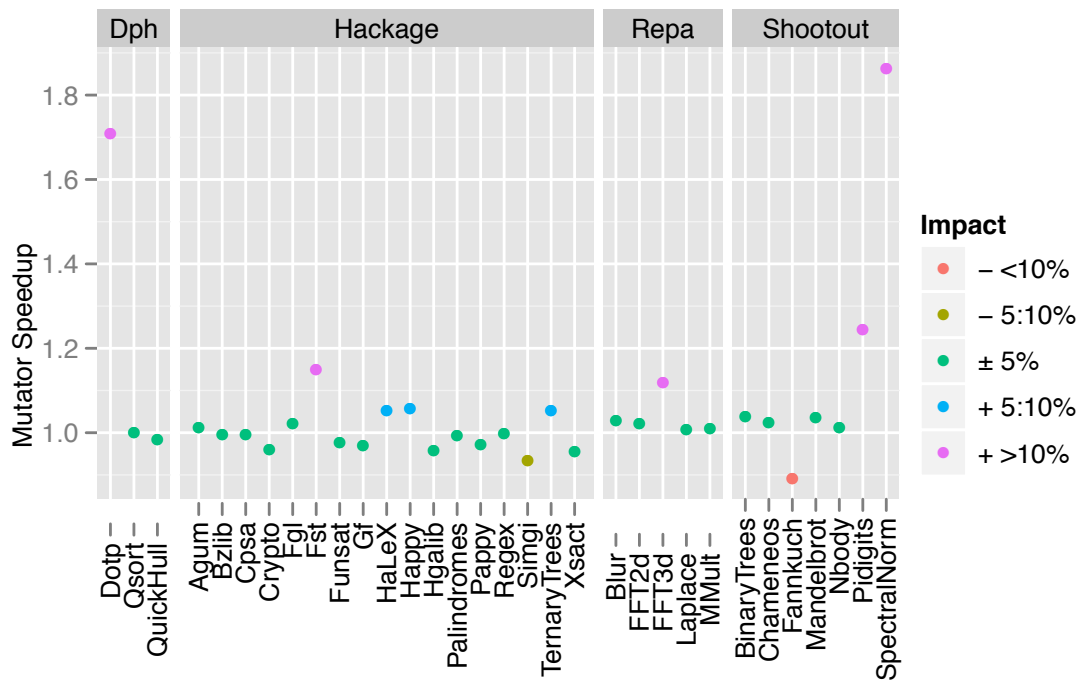


Figure 6.18: Performance of benchmarks under Htrace with a hotness threshold 100,000. The geometric mean speedup is 5%.

The improvement we get with different training and performance sets will depend on how much the control flow is driven by the program input.

Figure 6.18 shows the impact of running the Fibon benchmarks under Htrace. A hotness threshold of 100,000 was used to collect these results. Eight of the thirty two benchmarks (25%) show a speedup of 5% or greater, with five of those benchmarks showing a speedup of more than 10%. Only two of the benchmarks show a performance degradation of more than 5% with one benchmark (Fannkuch) showing a slowdown of 12%. The remaining benchmarks showed little difference in performance and were all within 5% of the non-Htrace version.

Htrace does a decent job at improving the performance of some benchmarks, but it is largely ineffective for others. To understand the performance numbers we need to take a detailed look at where the programs spend their execution time. Figure 6.19 shows the speedup numbers with the benchmarks grouped based on time spent on

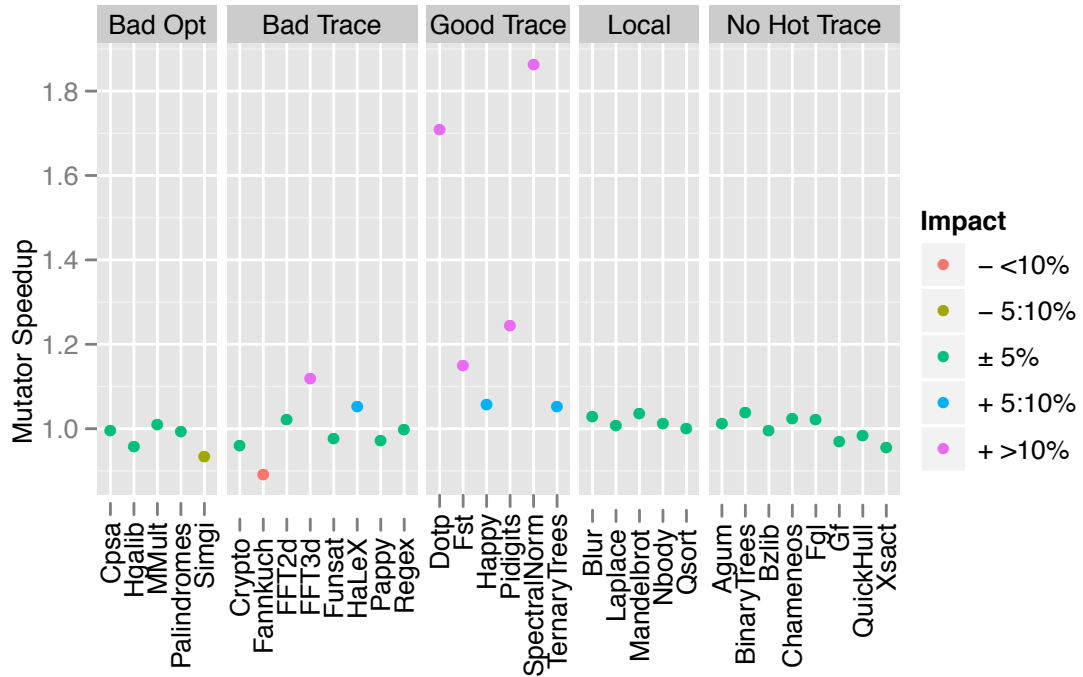


Figure 6.19: Performance of benchmarks under Htrace with a hotness threshold 100,000 grouped by disposition. The disposition categories are described in Table 6.2

Category	Description
Bad Optimization	Spends time on trace, but optimization is ineffective
Bad Trace	Spends time on a trace, but has poor completion rate
Good Trace	Spends time on trace and optimization is effective
Local Trace	Spends time on local (one function only) traces
No Hot Trace	Does not spend time on traces

Table 6.2: Benchmark disposition categories

traces and the completion rate of those traces.

The benchmarks are grouped into five categories based on profiling results. Each benchmark was run with a sampling-based profiler to get an understanding of where the program was spending its execution time. The dynamic profile was combined with the trace completion statistics gathered during shadow tracing to categorize the benchmarks according to their profiling and trace statistics. The different categories are listed in Table 6.2 and described below. Table 6.3 displays the data used to

categorize the benchmarks.

The *Bad Optimization* category is for benchmarks that spend a good amount of execution time on a trace we built, but do not get a performance improvement from the increased optimization scope. The performance of benchmarks in this category could possibly be improved by implementing some trace-specific or Haskell-specific optimizations. Most of the benchmarks in this category see little change in performance when running with Htrace. The Simgi benchmark is the exception, which has a slowdown of about 7%.

The *Bad Trace* category is for benchmarks that spend time on a trace, but have poor completion rates. These benchmarks generally do not get an advantage from Htrace because the trace heuristics do not find a good trace. The worst performing benchmark, Fannkuch, is in this group. It suffers greatly because it spends a lot of execution time on the trace, but often exits by the second function on the trace. On entry to the trace, the LLVM register allocator spills many of the registers to free up more registers for allocation in the loop body. These spills must be reloaded at the trace exit, and because the trace rarely completes the extra stores and loads add overhead which increases the running time of the benchmark.

Not all benchmarks suffer a performance penalty for having bad traces. The FFT3d benchmark achieves a speedup of 12% despite having a poor completion rate of its top trace. Although the completion rate of the top trace is only about 8%, the common trace exit points are at the bottom of the trace. Because the common exits are near the end of the trace the optimizations for a large part of the trace will still be effective. The result here points out that not just the trace completion rate is important for judging performance, but also the relative completion rate that measures how far we typically get through the trace before an early exit.

The *Good Trace* category is for the benchmarks that spend a good amount of execution time on a trace, and the trace also has a high completion rate. The com-

Bad Optimization					
Benchmark	Profile	Rank	Completion	Functions	Blocks
Crypto	13.8%	-1	86.97%	22	55
Hgalib	15.1%	1	100%	4	9
MMult	77.8%	1	65.04%	39	87
Palindromes	4.3%	-1	99.73%	3	7
Simgi	16.3%	1	92.65%	23	55
Bad Trace					
Benchmark	Profile	Rank	Completion	Functions	Blocks
Cpsa	13.0%	1	3.56%	20	52
Fannkuch	16.6%	1	0.22%	19	53
FFT2d	10.6%	1	0.09%	12	30
FFT3d	19.7%	1	8.04%	30	71
Funsat	2.2%	-1	0.15%	13	27
HaLeX	8.1%	4	3.64%	14	31
Pappy	1.5%	-1	0.03%	43	101
Regex	7.8%	2	15.27%	38	101
Local Trace					
Benchmark	Profile	Rank	Completion	Functions	Blocks
Blur	49.7%	1	97.19%	1	7
Laplace	68.8%	1	98.02%	1	8
Mandelbrot	80.4%	1	97.20%	1	4
Nbody	55.3%	1	66.67%	1	3
Qsort	3.1%	-1	60.97%	1	5
Good Trace					
Benchmark	Profile	Rank	Completion	Functions	Blocks
Dotp	34.9%	1	100%	4	10
Fst	51.9%	1	92.78%	13	27
Happy	21.4%	-1	95.12%	13	27
Pidigits	64.7%	1	95.66%	4	9
SpectralNorm	23.5%	1	99.92%	15	33
TernaryTrees	9.2%	-1	86.35%	12	28

Table 6.3: Disposition of benchmark traces. The *Profile* column lists the absolute percent of execution time the program spent on the top trace. The *Rank* column lists the position of the top trace in the overall profile results for that benchmark. A negative number indicates the position is relative to the first profile entry that does not come from the garbage collector. The *Completion* column lists the completion rate of the top trace. The last two columns give the size of the trace in functions and blocks.

bination of finding the right program traces and having effective optimizations make this the best performing category. All the benchmarks in this group have a speedup of more than 5%.

The *Local Trace* category contains all of the benchmarks that spend time on traces that span only a single function. The traces will be simple tail recursive functions, which LLVM can turn into a loop. The benchmarks in this group will not show a great performance improvement because LLVM can already see and optimize the loop. The Qsort benchmark is unique in this group because it does not get a big speedup from the LLVM backend (see Figure 2.4). The reason for the lack of speedup on Qsort is because its execution time is spread across many small traces. There is no single piece of code that dominates execution time and is amenable to optimizations.

Finally, the *No Hot Trace* category identifies the benchmarks that spend no time on the traces we build. These benchmarks either have no traces that are hot enough to show up in a profile, or we could not identify traces for them at all (e.g. Bzlib, Chameneos). We see no real benefit or detriment in these benchmarks because our tracing scheme has no impact on the code that actually executes. These programs are unlikely to benefit from trace-based optimization because of their lack of good hot traces.

Table 6.3 shows the detailed data used to categorize the benchmarks. The *No Hot Trace* category is omitted from the table because the trace data is irrelevant since there were no hot traces in those benchmarks. The *Bad Optimization* group has two distinct subgroups. The Hgalib and Palindromes benchmarks have small loop traces that we would expect LLVM to be able to optimize. The traces from the other benchmarks are much larger and it is likely that LLVM has trouble optimizing the function that results from inlining all of the constituent trace functions. We could probably improve the performance of the *Bad Optimization* benchmarks that have large traces by improving our trace builder to prune away the cold paths in

the cloned function. The *Bad Trace* group contains benchmarks with top traces that are generally quite large. The large trace size combined with a poor completion rate generally eliminates the performance benefits we might hope to see. The *Good Trace* benchmarks have top traces that are on the smaller side. The smaller trace size and high completion rate provides the opportunities for LLVM’s optimizations to improve performance.

To understand why we get good performance on the *Good Trace* benchmarks we can take a detailed look at what happens in the *Dotp* benchmark. *Dotp* is a nice example because the traces are fairly small and easy to understand what is happening. The majority of the time is spent on two traces, one of which is local. We will discuss the non-local trace because that is the source of all improvements and it is also the function where the benchmark spends the most time.

Figure 6.20 shows the trace control flow in the original code compared to the control flow in the restructured code. The hot trace occurs at a point where we are copying data from one array to another using a modular index variable. The entry to the trace saves the live variables to the stack and calls the `mod` function passing an index variable. At the return from the `mod` function we reload the index variables from the stack and use them along with the value returned from `mod` to perform the copy. Finally, we updated the index variables and call back to the trace entry function.

After restructuring the code around the trace, we have a single function with a loop that performs the copy. Table 6.4 lists five major benefits from the increased scope of our trace function. One of the most important benefits is that the register allocator can allocate the index variables to registers which eliminates one reload from the stack on every iteration. In the original function, the updated index values computed at the bottom of the loop in the `f_ret` function are stored and reloaded from the stack. In the trace function we only store the updated values to the stack and then keep them in registers until they are needed again at the bottom of the

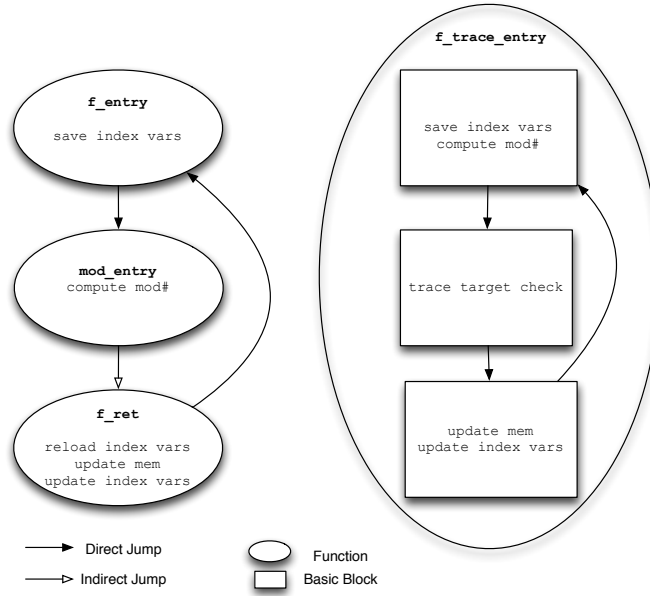


Figure 6.20: Trace code shape for the Dotp benchmark. The original code (Left) includes three separate functions, one of which is reached by an indirect jump. The restructured code (Right) is contained in a single function. The improved scope allows the register allocator to allocate the index variables to registers across the computation of `mod`.

- | |
|--|
| <ol style="list-style-type: none"> 1. Register allocation of variables around loops 2. Reduced stack pointer manipulations 3. Hoisted loop invariant code 4. Compact loop code 5. Improved instruction scheduling |
|--|

Table 6.4: Sources of improvement for restructured low-level code

loop. The original code could not see through the `mod` function call and so it had to reload them on each entry to the `f_ret` function. The improved register allocation in the trace function saves four extra loads per iteration.

In addition to the register allocation benefits, we see several other improvements. Collapsing the control flow into a single function reduces the number of stack pointer manipulations. In the original version the stack pointer will be incremented and decremented on each iteration of the loop. In the combined code, we only have one increment and decrement for all the iterations. We also see opportunities for moving loop invariant code out of the loop. In the Dotp benchmark, the stack limit and

heap limits can be computed once and stored in a register. We would prefer entirely eliminating the stack check in the loop body, but currently LLVM does not hoist the stack check control flow. Collecting the code into a single function also improves the locality for the instruction cache. In the original version of Dotp, the call to `mod` will jump to distant code. The improved version keeps all of the code close together and will have better cache behavior. Finally, the traced code has more opportunity for better instruction scheduling. Tracing through the `mod` function provides many more arithmetic instructions that can be scheduled in the latency of the memory operations. These extra instructions can essentially execute for free in the traced version.

Although we see a great performance gains in the Dotp benchmark, there is still much room for improvement in the low-level code. The greatest source of overhead that still remains is the stores and loads of values to the Haskell stack even after the tail call elimination optimization has run. There are two issues that cause the stores to remain. The first issue is that the stores to the Haskell stack look like writes to memory so LLVM is unable to remove them because it does not realize the stores are dead after the function call returns. The second issue is that LLVM is not moving the stores out of the loop down to the trace exit paths. The stores are not entirely dead since they may be used on paths off the trace. We would like LLVM to move the stores off the hot path, but it is unable to prove that it is safe to do so. Teaching LLVM about the Haskell stack would help improve the code generated for the restructured code.

6.3.2 Trace Statistics

In this section we examine several characteristics of the traces found by Htrace. We first look at the number of traces found and compare it to the number found by DynamoRIO. Next we look at the number of broken traces and how increasing the

Quartile	Number of Traces
Q_1	4.75
$Q_2 = \text{Median}$	12.50
Q_3	33.00

Table 6.5: Number of traces found by Htrace. The median number of traces (e.g. Q_2) found for the Fibon benchmarks is 12.5. Compare to Table 5.3 which shows a much higher number of traces found by DynamoRIO.

scope available to Htrace impacts the broken traces count. Finally, we examine the completion rate of traces.

Number of Traces

The number of traces found by Htrace is shown in Figure 6.21. We can see that there is considerable variation in the number of traces found for each benchmark. The distribution of trace counts is summarized in Table 6.5. Compared to DynamoRIO, the number of traces found for each benchmark is quite small. DynamoRIO found a median of 634 traces for each benchmark, compared to the 12.5 found by Htrace. The small number of traces found for most benchmarks indicates that the tracing heuristics used by Htrace are better suited for Haskell.

The Shootout, Dph, and Repa benchmarks typically have fewer traces than the Hackage benchmarks. The Hackage benchmarks are larger programs, so it makes sense that they would have a greater number of hot traces than the smaller programs from the other groups. The major exception is Qsort, which has the most traces of any benchmark. The large number of traces in Qsort are because the code path is highly dependent on the data (because it is a sorting routine) which leads to a large number of frequently executed trace headers since the data is driving the trace down different execution paths.

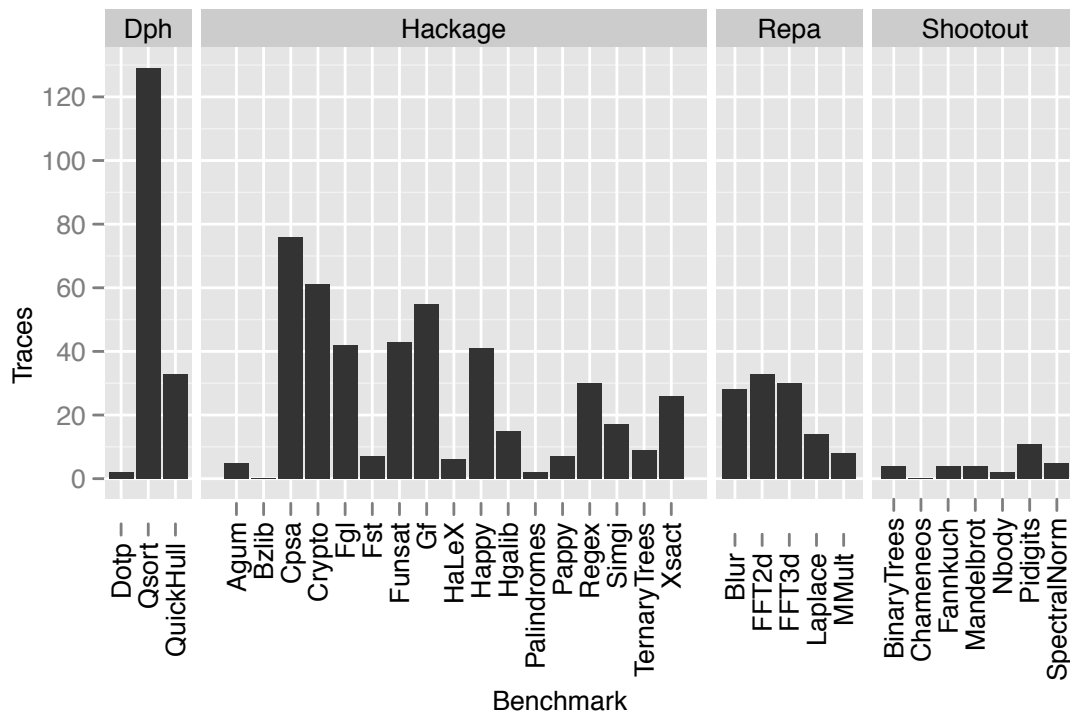


Figure 6.21: Number of traces found by Htrace with a hotness threshold of 100,000.

Trace Types

Figure 6.22 shows the different types of traces found by Htrace. The traces are categorized into three types: Loop, Long, and Local. Loop traces start and end at the same block and include at least one other block in between. Long traces are the traces that we stop recording because they have reached the length limit. Local traces are the traces that contain only a single block.

The trace types show a distinct trend among the Hackage group. These benchmarks tend to have a greater number of Long traces. We can attribute this difference to the non loop-based nature of these benchmarks. The Hackage benchmarks are more likely to have complex control flow which results in the larger number of Long traces. Both the Dph and Repa benchmarks have a larger number of Local traces. These Local traces are actually showcasing the ability of GHC to manifest program loops as tail recursive functions. We would expect to see more Local traces in these groups

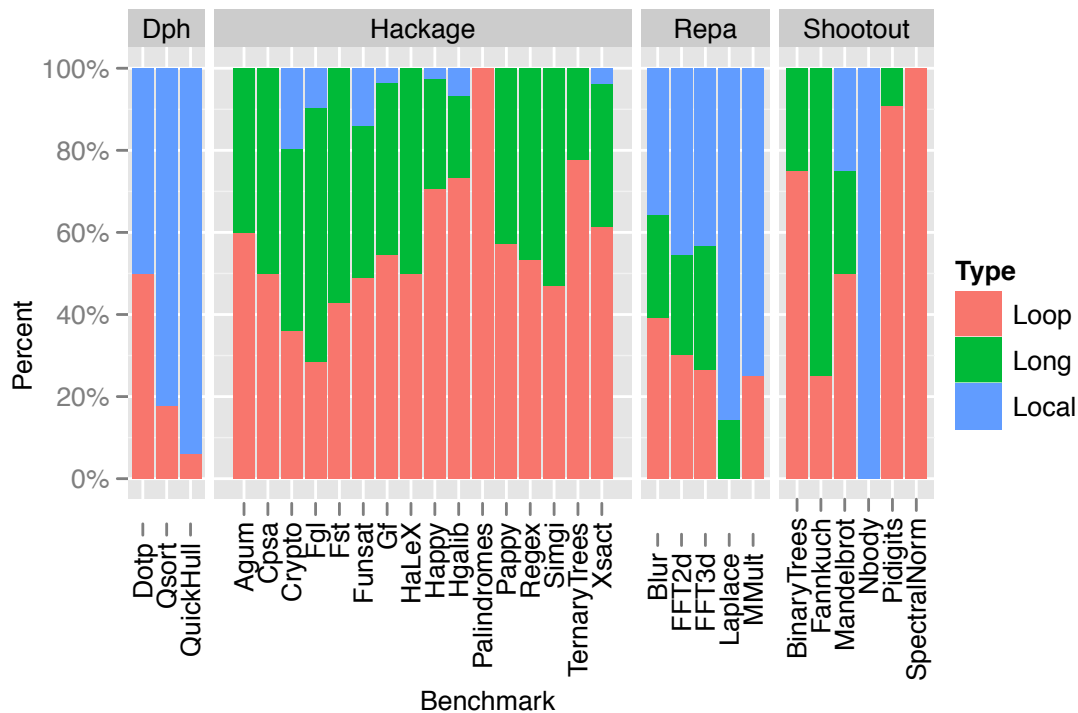


Figure 6.22: Types of traces found by Htrace with a hotness threshold of 100,000.

since they tend to be loop-based benchmarks. The distribution of trace types we have examined so far has been over all the traces found by Htrace. We can also look at the distribution of trace types weighted by the execution frequency.

Figure 6.23 shows the distribution of trace types weighted by the number of entries. The trace entry counts were collected by the shadow tracing described in Section 6.1. Although the entry counts do not necessarily correspond to execution time, we can get a sense of the relative frequency of what kind of traces are entered most frequently.

The major change that we see is the large reduction in the percent of Long traces. The reduction occurs across all the benchmark groups. The Long traces are nearly eliminated from the Repa and Shootout groups. The Hackage group still has a few benchmarks with a large percent of Long traces.

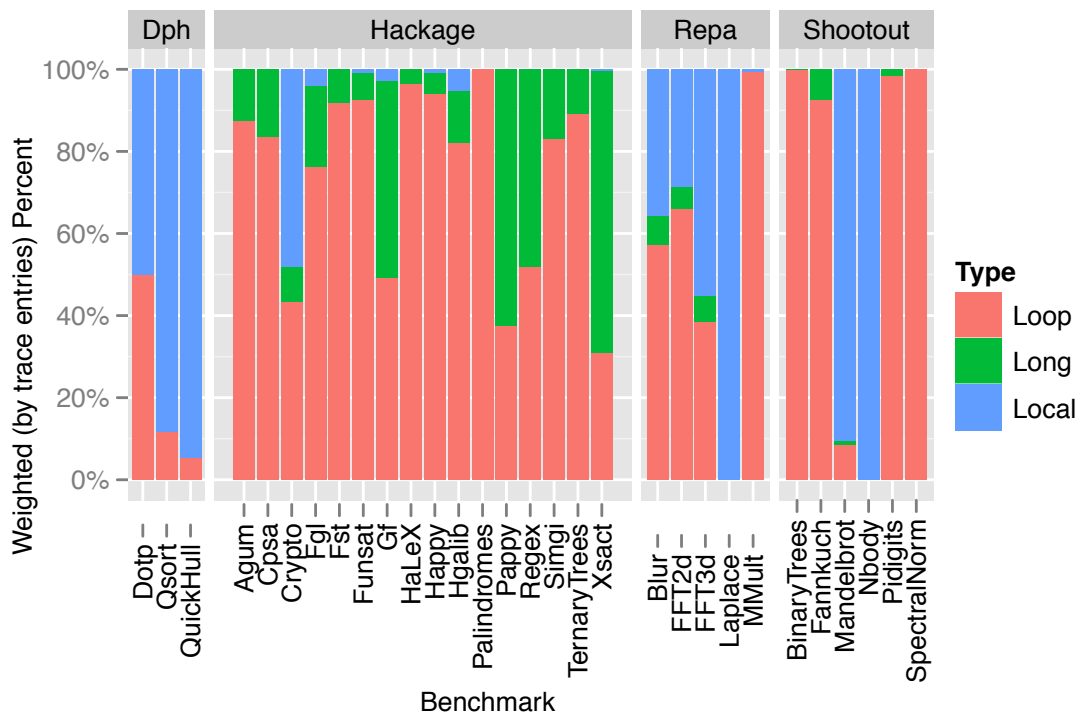


Figure 6.23: Types of traces found by Htrace weighted by number of trace entries. Hotness threshold is 100,000.

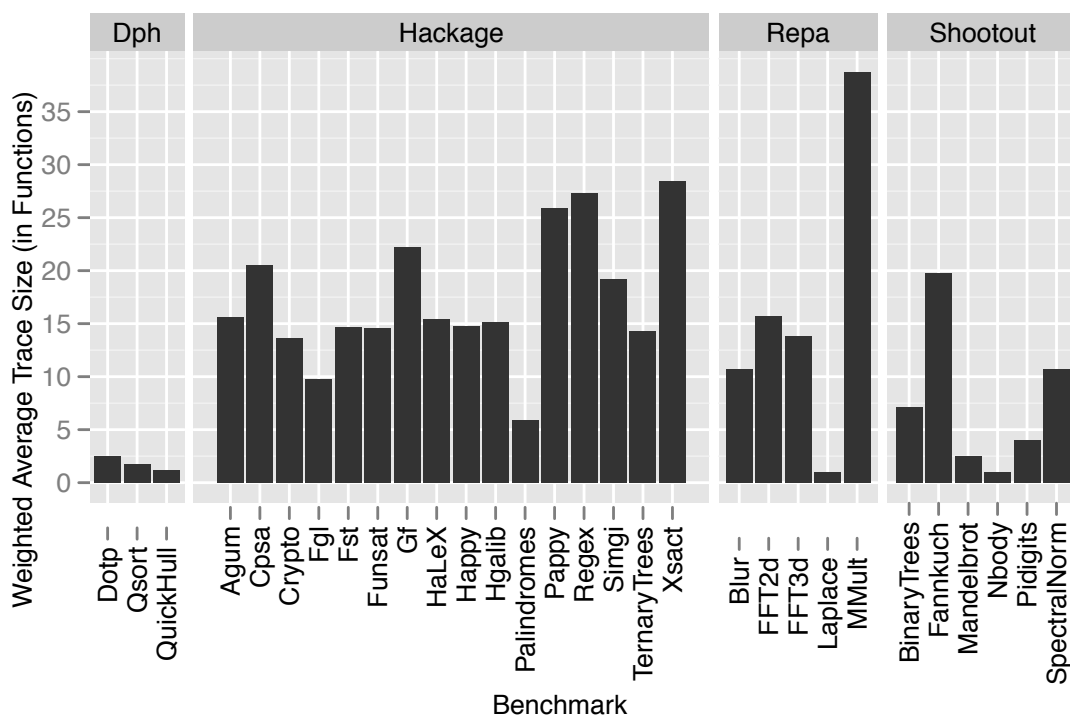


Figure 6.24: Length of traces found by Htrace with a hotness threshold of 100,000. The average weighted trace size is 13.6 functions with a standard deviation of 9.3.

Trace Length

Figure 6.24 shows the average length of traces found by Htrace. The average is a weighted average where the weights are the trace entry counts recorded by the shadow tracing technique. The length of the trace is measured in the number of different functions that appear along the trace.

The trace lengths vary quite a bit across the benchmarks. The Dph group tends to have smaller traces compared to the other groups. The Hackage benchmarks tend to have large average trace lengths. The longest average length is the MMult benchmark where the average trace contains 38 functions. The large size of the MMult trace points to a reason why it is not gaining much benefit from Htrace. Htrace restructures the program around a trace by inlining the functions on the trace, and it is likely that the standard LLVM optimizer is overwhelmed when trying to optimize

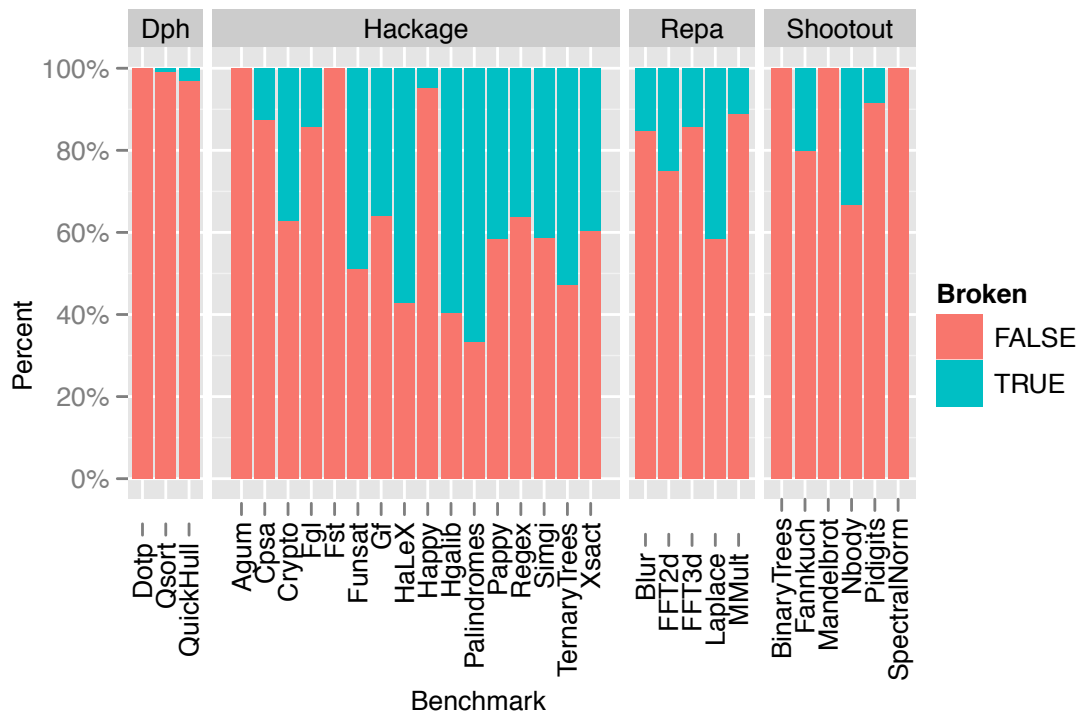


Figure 6.25: Percent of broken traces found by Htrace with a hotness threshold of 100,000.

a single function that has been constructed from 38 individual functions which are inlined at their call sites.

Overall the size of the traces is encouraging. The heuristics used by Htrace show that we can recover scope an optimization scope that spans a large number of functions. The same scope would be hard for an optimizer to discover statically.

Broken Traces

Figure 6.25 shows the number of broken traces found by Htrace. As described in Section 6.1 a broken trace is a trace that jumps to code that we cannot see. The external code will be either a Haskell library for which we do not have source code, the GHC runtime library, or an external C library.

We can see that the Hackage benchmarks contain the largest number of broken traces. These broken traces are largely from traces that enter Haskell libraries that we

did not include in the standard library files we copy for tracing purposes as described in Section 6.2.3. For example, we do not trace through any code for performing I/O. The remaining benchmark groups tend to have a small number of broken traces. These results suggest that we can capture most traces by including a small set of library and runtime files. The Hackage benchmarks stand to benefit most from increasing the number of library files that we include for tracing purposes.

Figure 6.26 shows a detailed look at how the tracing scope effects the number of broken traces. The scope is broken down into the P, PR, and PRL levels. The different scopes were constructed by copying different amount of LLVM bitcode files to the Htrace directory used for collecting program traces. The P scope copies only the bitcode files for the program source code. The PR scope copies both program code and the GHC runtime files listed in Figure 6.10. The PRL scope includes the same program and runtime files, and it also includes all of the library files listed in Figure 6.17.

The results in Figure 6.26 were collected using a hotness threshold of 10. The different threshold means that the broken percent will not match that found in Figure 6.25. The main trend we want to check is that increasing the scope reduces the number of broken traces. For the most part, when we increase the scope we do see an increase in non-broken traces. It can also be the case that increasing the scope reduces the percent of non-broken traces. The percent of non-broken traces will decrease if we find a lot of broken traces in the newly available scope. Although a few benchmarks exhibit a decreasing percent of non-broken traces, the general trend is the increased non-broken rate that we would expect.

Overheads

Table 6.6 shows several overheads associated with Htrace along with several raw size measurements of the storage needed for the profiling data and bitcode files.

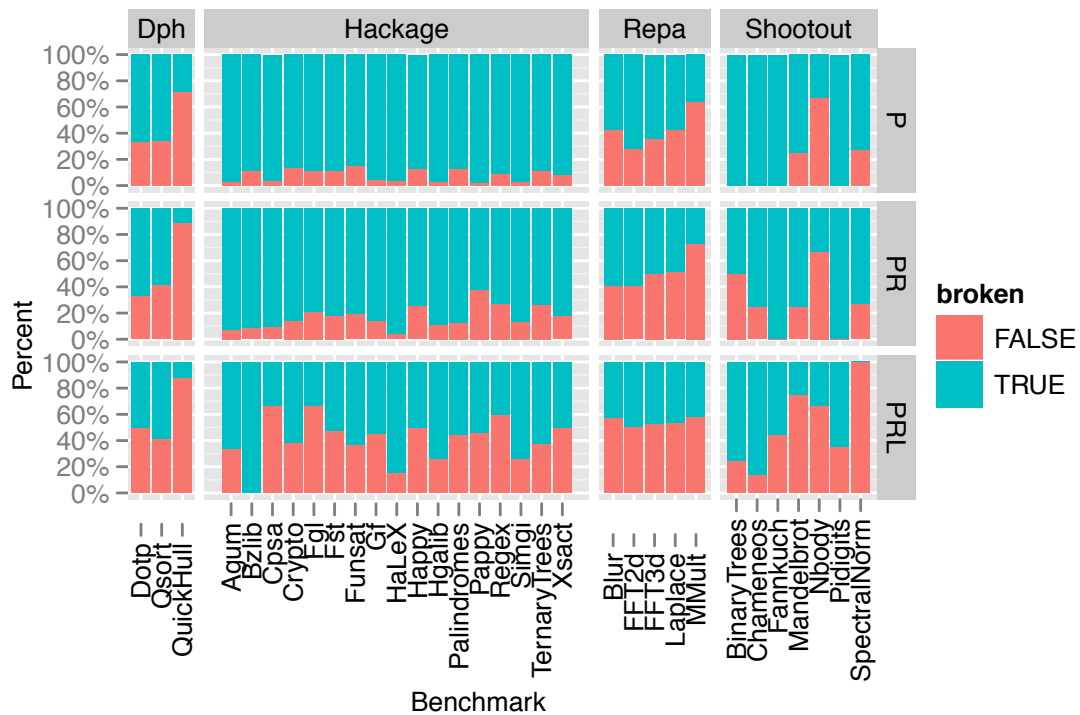


Figure 6.26: Effect of trace scope on percent of broken traces found by Htrace. The data from this graph uses a hotness threshold of 10. The difference scopes are denoted by the P, PR, and PRL labels. The P scope is program source code only. The PR scope is source code and runtime files. The PRL scope allows tracing through the source code, runtime and library routines.

	Minimum	Average	Maximum
Profiling Time Overhead	1.4×	7.68×	116.9×
Bitcode Space Overhead	1.0×	1.02×	1.09×
Baseline Bitcode Size	15.33 MB	21.55 MB	68.85 MB
External Trace Size	92 Bytes	27.69 KB	111.70 KB

Table 6.6: Htrace overheads and file sizes. The Profiling Time Overhead measures the slowdown we see when collecting the profile data to find program traces. It compares against a normal non-profiled execution time. The Bitcode Space Overhead measures the code expansion that comes from cloning functions to build traces. It compares to the Baseline Bitcode Size that comes from combining the original LLVM bitcode files for the program, library, and runtime bitcode. The External Trace Size shows the size of the profile data written out by the Trace Runtime.

The Profiling Time Overhead shows the slowdown we see by running the program through Htrace to find program traces. The overhead comes from interpreting the bitcode files using `lli` and the callbacks to the Trace Runtime that are used to find and record program traces. The average profiling time overhead is 7.7×, with a maximum slowdown of 117× on the Dotp benchmark. Although the overhead may seem large, it only occurs in the profiling step. Once the traces are found and written to the external format, they can be built and optimized without incurring further time penalties in the compilation process.

The Bitcode Space Overhead shows the increase in the size of the bitcode due to trace instantiation. The increase is measured against the baseline bitcode size that comes from combining the bitcode files for the program, library, and runtime bitcode as described in Section 6.2.3. In most cases the overhead is very small with a maximum increase of 9%. Finally, the External Trace Size is the size of the profiling data written by the Trace Runtime after finding traces. The external trace files are very small with the largest file measuring only 112 kilobytes.

Trace Completion Rate

Figure 6.27 shows the weighted average completion rate of the traces found by Htrace. The completion rates for individual traces were gathered during the shadow tracing

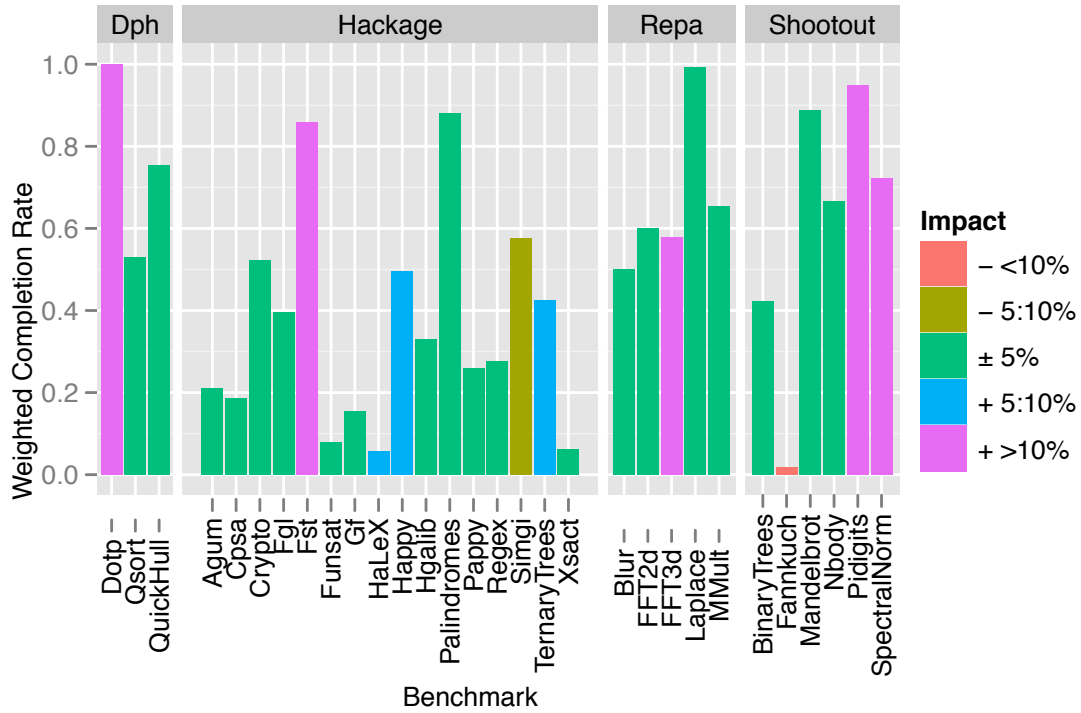


Figure 6.27: Weighted trace completion rate of traces found by Htrace. The average trace completion rate is weighted by the relative execution frequency of the trace for a given benchmark. The Impact percent is the performance impact from Figure 6.18. The average completion rate is 50% with a standard deviation of 30%.

portion of the trace finding technique used by Htrace. A trace is considered to run to completion if it reaches the last basic block in the trace. The average completion rate is computed as a weighted average where the completion rates of individual traces are weighted by the number of times that trace was entered.

The completion rates shown in Figure 6.27 are shaded according to their performance impact from Figure 6.18. We shade the completion rates by performance data to see how completion rate impacts performance. The best performing benchmarks generally have a high completion rate, although a high completion rate is not a guarantee of success. The Laplace, Mandelbrot, and Nbody benchmarks have a high completion rate because they have a large proportion of Local traces, which do not translate into a performance improvement through Htrace. The Palindromes

benchmark also has a very high completion rate, but it does not gain a performance boost because LLVM is unable to effectively optimize the restructured code. The Fannkuch benchmark is the worst performing benchmark and it has the lowest completion rate. The performance degradation we see on that benchmark is because of the extra overhead introduced by constantly exiting early from its traces.

The trace completion rate varies according to the path that was recorded when the trace header became hot. If we record the trace too early or too late, then we might record a non-representative path from that trace head. Since trace completion rate can have an effect on performance, we want to choose a good hotness threshold for when to record a trace. Next we examine the effect of modifying the threshold for considering when a trace is hot.

6.3.3 The Effect of Hotness Thresholds

The hotness threshold dictates the number of times a trace header will be entered before we record a trace starting from that block. The hotness threshold should be set to a level that allows us to record a trace that represents the most common path from the trace header.

The hotness threshold has two primary effects. First, it has a direct effect on the number of traces we find. A low threshold will start tracing much earlier and as a result will tend to find many more traces. However, since we only start tracing on a header that is not already part of a trace a low threshold may cause us to find fewer traces by covering other potential trace heads with an early trace through them. The second effect the threshold has is to change the trace paths found by Htrace. A program may have distinct phases and different thresholds will capture different traces according to the current program phase.

In this section we measure the effect that the hotness threshold has on the traces we find. Our results show that Haskell programs generally do better with a higher

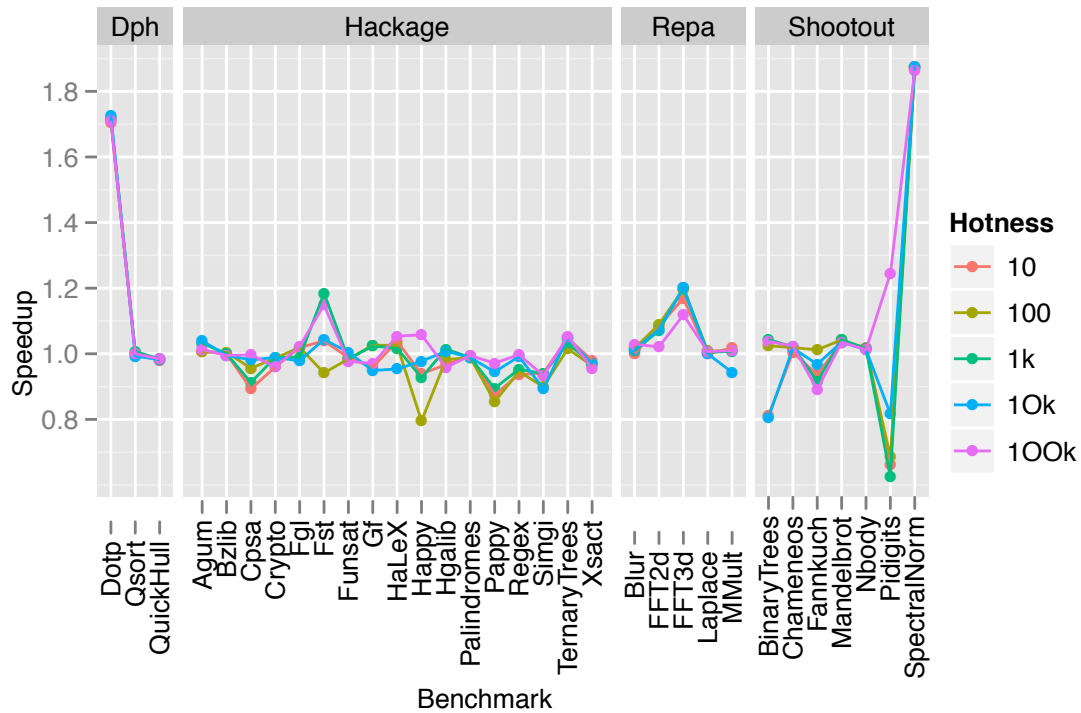


Figure 6.28: Fibon speedup by hotness threshold.

threshold, although the best threshold varies quite a bit among programs.

Speedup by Hotness Threshold

Figure 6.28 shows the speedup of the Fibon benchmarks for various hotness thresholds. There appears to be considerable variation in the benchmark performance for different thresholds. Among the Dph and Repa benchmarks, the thresholds seem to have less of an effect. The reduced effect is likely because these benchmarks have more local traces and because the hot execution paths tend to be loops that do not have great variation in control flow. The Hackage and Shootout benchmarks tend to show a greater response to the change in hotness threshold. These benchmarks are more control-oriented, and the different hotness thresholds pick out different traces.

The Pidigits benchmark in particular has a dramatic response to the hotness threshold. It goes from the worst slowdown of nearly 67% to one of the best perform-

ing benchmarks. The reason for the change is that the lower thresholds are not finding the hot path in the benchmark and the traces are exited early. These early exits cause the degradation in performance. The benchmark uses multiprecision arithmetic to compute the digits of π . The multi-precision integers are represented by an algebraic data type that has cases for word sized integers and greater-than-word-sized integers. Starting the traces too early results in traces going through the case for smaller integers while the majority of the execution time is spent in the case for larger integers.

Distribution of Best Hotness Thresholds

Figure 6.29 shows the distribution of the best hotness thresholds across all benchmarks. In general, the larger hotness thresholds work better for most benchmarks. The reason that larger thresholds work better is twofold. First, the higher thresholds produce fewer spurious traces that simply add overhead. Eliminating these traces allows for more productive traces to be built. Second, the larger thresholds allow traces to occur along the common paths, which take some time to manifest.

The high threshold hotness threshold is quite a contrast to the results reported by [Duesterwald and Bala \[2000\]](#) for the Dynamo trace-based optimizer. They found that a small threshold of 50 produced good traces. The differences are likely due to lazy evaluation obscuring the common path at low thresholds and the fact that Dynamo is tracing at runtime, which means they need to quickly find the traces. Our offline tracing system can be more patient and wait for the hot path to develop without losing any performance because of time spent not executing on a trace. Another potential reason for the difference our hotness levels and those reported by Duesterwald and Bala is the nature of the codes under study. They experimented primarily with C/C++ codes from SPECint. Our Fibon benchmarks should capture a more diverse set of program behaviors, and these behaviors might suggest a need for larger hotness thresholds.

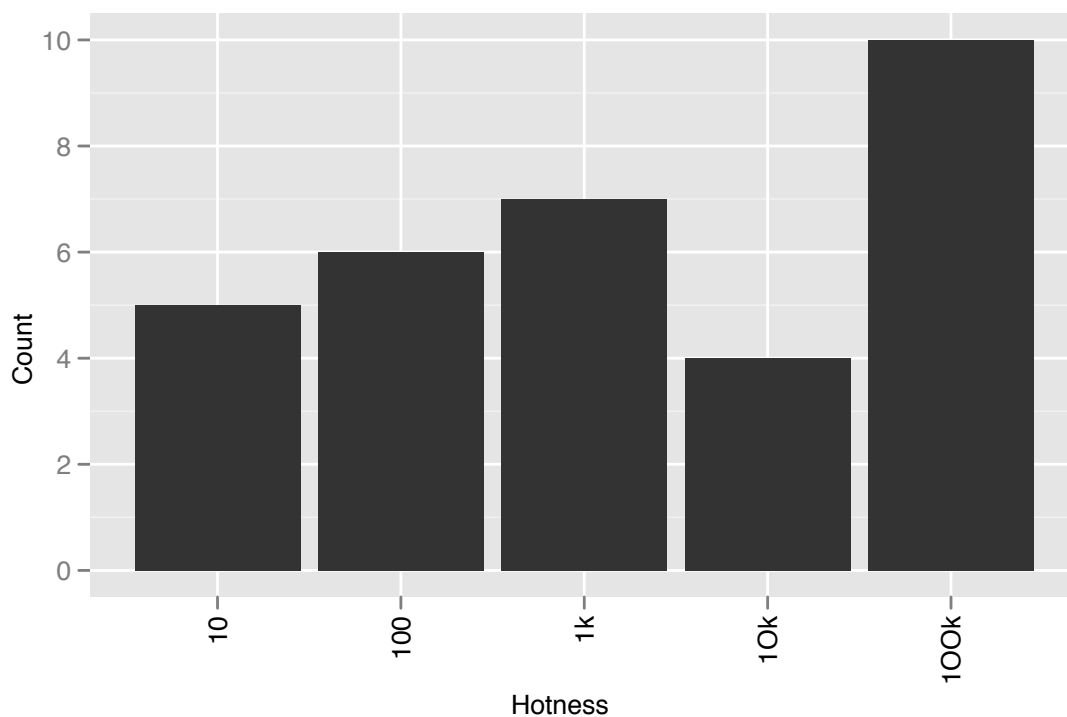


Figure 6.29: Distribution of hotness thresholds for best speedup.

Best Speedup by Hotness Threshold

Figure 6.30 shows the speedup of each benchmark using the best hotness threshold. We can see that there is no clear winner as to the best threshold. Compared to the fixed threshold results in Figure 6.18, we have added FFT2d to the benchmarks that improve by more than 5%. The geometric mean speedup is about 7% across all benchmarks. Among the benchmarks that improve by more than 5%, 8 out of 9 work best with the hotness threshold of 1000 or greater.

The great variety of hotness thresholds points to a potential area for improvement in our tracing scheme. Currently we use a fixed hotness threshold and then record a single trace after the threshold is met. We could take advantage of the shadow tracing to actually build alternate traces when the exit paths of the original trace become hot. These alternate traces may do a better job of covering the actual hot path.

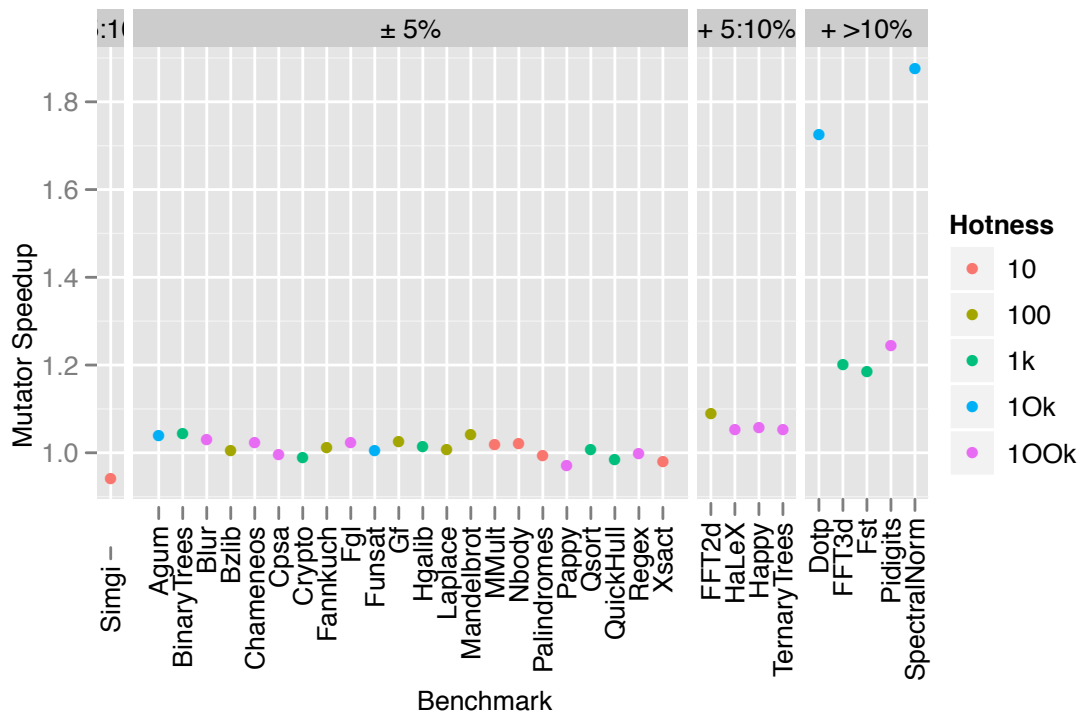


Figure 6.30: Best speedup for any hotness threshold.

Our static approach to tracing will have problems if the performance variety due to hotness thresholds is caused by actual phase changes in the behavior of the application. In this case, the runtime trace systems have an advantage because they can respond to the phase change by throwing out the current trace and building a new one. Static systems like as Htrace will have more difficulty coping with such a phase change.

6.4 Conclusion

In this section we described the design and implementation of Htrace, a system for optimizing Haskell programs by restricting the code around program traces. We found that we could achieve a maximum speedup of 86% and an average speedup of 5% across 32 benchmarks by running low-level optimizations on Haskell code restructured

Parameter	Setting	Explored
Trace header policy	Function/thunk entry points	No
Allow headers in libraries	Yes	No
Hotness threshold setting	100,000	Yes
Hotness threshold granularity	Coarse	No
Trace entry replacement	Global	No
Trace scope	Program & Runtime & Libraries	Yes
Trace phase detection	None	No
Instantiate non-loop traces	Yes	No
Allow headers in trace	Multiple trace bodies & one root	No

Table 6.7: Trace parameters used in the Htrace design.

to take advantage of the program traces. Our best results are produced for programs that contain frequently executed loops that span a number of functions. When the program is restructured around the program traces the loop is visible to the compiler which makes it possible to perform beneficial optimizations such as register allocation and invariant code motion.

The design space for building a system like Htrace is quite large. We identified a number of deliberate choices that must be made when designing such as system. Table 6.7 lists the parameters we considered when building Htrace. There is a considerable amount of room for future exploration of the design space. A few ideas are sketched below.

Trace header policy Our current policy is to mark all function and thunk entry points as potential trace headers. We could use many alternate policies including the use of profiling data to find hot candidates and allowing trace headers on the `ret` function return points generated for lazy evaluation continuation points.

Allow headers in libraries Currently we mark potential trace headers in all libraries. Alternately, we could limit trace headers to the program source code or a subset of the libraries.

Hotness threshold setting We set the threshold to 100,000 entries to a trace header before we start recording. The limit was chosen after experimenting with a variety of thresholds. We saw that the best threshold varied across the benchmarks and we could modify the design to try and take advantage of that fact. One possibility is to use a more heavyweight profiling algorithm such as [Ball and Larus \[1996\]](#) to find potential trace paths.

Hotness threshold granularity The hotness threshold is uniform across all trace headers. We could set the threshold separately depending on the location of the trace header. For example, we could set the threshold higher for trace headers in frequently used library code.

Trace entry replacement After instantiating a trace by inlining all of its component functions we replace all uses of the trace header function with the newly created trace. An alternate policy would be to selectively replace the header. We could do a selective replacement by keeping track of the locations where the trace is commonly entered and replacing only the uses at those locations.

Trace scope We explored varying the trace scope over the program, runtime, and library code. The libraries were limited to a hand-selected set of frequently used libraries. We may be able to improve performance by further widening or narrowing the selected set of library codes.

Trace phase detection Once a trace is built we no longer start new traces from that header. If the program exhibits different behavior in different phases our trace will not capture these changes. We can take advantage of the shadow tracing to try and detect phase changes and build new side traces to take advantage of the alternate exit paths leading from the trace.

Instantiate non-loop traces Currently we instantiate non-loop traces. It may be

better to ignore the traces and only instantiate the loop traces.

Allow headers in trace When we build a trace we include both header and non-header blocks. Once a header block is included in a trace it will not be the root of a new trace. There are a couple of different policies that can be used here, including stopping a trace when it hits a trace header and allowing a header to both be a trace and act as a trace root.

As we can see there are many choices we can make about how to find and build traces in Haskell programs. Exploring these tradeoffs and alternate design points is an interesting area of future work.

Our current design choices allowed us to explore the opportunities of trace-based optimization for Haskell. The results are promising and can serve as a foundation for future explorations. The primary areas for future research are refining the trace building mechanism and building low-level optimizations specifically tailored to the Haskell program traces.

Chapter 7

Related Work

Our work draws connections to three related areas: static optimization of Haskell, dynamic optimization techniques, and feedback directed optimization. Each of these areas has its own rich history.

Static optimization of Haskell has been researched for many years and has yielded great advances in runtime performance. While this thesis does not focus on static Haskell optimizations, the choices made in these high-level optimizations greatly affects the low-level code whose optimization is the focus of our work.

Dynamic optimizers run compiler optimizations at runtime instead of at compile time. They are able to take advantage of additional runtime context to improve the effectiveness of the optimizations. There are two main flavors of dynamic optimization: virtual machine and binary based optimization.

Dynamic optimization in virtual machines has gained a lot of favor in recent years with the widespread adoption of managed runtime languages such as Java and C#. While we are not working in the context of a virtual machine, our work can benefit from some of the same techniques used in these virtual machines. Dynamic binary optimization is a more recent addition to the compiler optimization arsenal. It has yet to see wide scale adoption in commercial compilers and there are still many

opportunities to make significant contributions to this young field. These works are more closely related to the work in this thesis because we have experimented with DynamoRIO, which is a binary-based optimization framework.

Feedback directed optimization (FDO) ¹ are techniques that combine both dynamic and static approaches. FDO uses a separate profiling run to collect information about the runtime behavior of the application. It then uses the profiling data to statically optimize the program. The optimization is done completely ahead of time so that it incurs no runtime penalty. The optimizations use the profile to take advantage of dynamic information without any runtime overhead. The disadvantage of FDO compared to dynamic optimization is that FDO relies on a fixed set of program inputs so it has more difficulty responding to changes in program behavior. Htrace is an example of a feedback directed optimization framework.

7.1 Static Haskell Optimization

The Haskell language was created out of the desire to have a common lazy functional programming language to focus the efforts of the community [Hudak et al., 2007]. The first Haskell standard was published in 1990 and the standardization process continues to publish official accounts of the language. The latest version at the time of this writing is the Haskell 2010 standard. Although the Haskell language itself is standardized by a committee, the actual implementation of the language can vary considerably from compiler to compiler. In this work we focus mainly on the implementation techniques used in the GHC compiler [Peyton Jones and Marlow, 2011], which is the fastest and most widely used implementation of Haskell currently available. For a general overview of the issues with implementing lazy functional languages, see the books by Peyton Jones [1987] and Peyton Jones and Lester [1992].

The seminal paper that describes GHC's strategy for implementing Haskell is

¹We consider FDO and profile guided optimization (PGO) as two terms for the same concept.

[Peyton Jones \[1992\]](#). In that paper Peyton Jones describes the *Spineless Tagless G-Machine* (STG machine) which is an abstract machine for implementing Haskell and shows how the STG machine can be implemented on standard hardware. Although the paper was written in 1992, the approach described is still used in GHC with the major exceptions being the handling of unknown function application [[Marlow and Peyton Jones, 2006](#)] (now `eval/apply` instead of `push/eval`) and the tagging of pointers [[Marlow et al., 2007](#)].

Haskell is implemented in GHC by first translating it to a core language (often just called *core*) which is a small functional language. Many transformations are performed on this representation that take advantage of the fact that core is a functional language. The core language is then translated to the STG language. The STG language is similar to core except that it has an explicit operational semantics which describes how it can be evaluated. The STG language is then translated to CMM, which is a low-level assembly-like language which can be fed to a native code generator.

The STG machine provides the framework for executing Haskell programs and has a strong influence over the final machine code that is produced. It dictates the opportunities that we can expect to find when performing the low-level optimizations described in this thesis.

One important performance improving transformation for Haskell that is performed on the core language is changing accesses of boxed data into accesses of unboxed data. A boxed data type is stored in the heap and is passed to functions as a pointer to the data. An unboxed value does not have this extra layer of indirection and can be accessed directly.

[Peyton Jones and Launchbury \[1991\]](#) describe a technique for handling unboxed data types in a Haskell compiler. They add unboxed data as separate types in the core language to allow them to transform the code to directly pass the unboxed val-

ues. One of the most important transformations they describe is the *worker/wrapper* transformation that changes a recursive function operating on boxed data into two separate functions: a worker that operates on unboxed data and a wrapper that takes the boxed data, unboxes it and then calls the worker function.

Unboxed values and the worker/wrapper transformation can greatly affect what the generated machine code looks like. For example, a tail recursive worker function that passes unboxed values can be translated to machine code that resembles a loop in a traditional language.

The worker/wrapper transformation works best on functions that always evaluate their arguments so that the values in the worker function can be passed around in evaluated form. *Strictness analysis* computes (an approximation of) which arguments to a function are always evaluated and thus which functions can be subjected to the worker/wrapper transformation. [Peyton Jones and Santos \[1998, Section 6.3\]](#) provide a description of a simple implementation of strictness analysis.

The worker/wrapper transformation is just one example of a transformation that can be performed on the core code of a Haskell program. [Peyton Jones and Santos \[1998\]](#) describe an approach to optimizing Haskell that they call a “transformation-based optimizer”. They advocate using repeated application of a large number of small transformations to optimize the Haskell core code. Their transformations encompass a number of traditional compiler optimizations such as function inlining, dead code elimination, and code motion. One of their most relevant findings for our work is that cross-module optimization is very important and they go to great lengths to be able to inline functions from other modules. Even though GHC inlines functions across modules, we found that we still need to build program traces through functions coming from multiple modules (e.g. from the standard libraries) in order to archive the most benefit from low-level optimizations.

While a Haskell compiler may contain many transformations, there will always be

some transformations that a programmer can prove are safe that the compiler cannot. [Peyton Jones et al. \[2001\]](#) describe a method that allows a programmer to specify “rewrite rules” that the compiler can use to automatically transform the program. The rules can be used to implement domain specific optimizations along with traditional optimizations such as specializing functions for a given type and specializing functions for evaluated arguments so that the worker/wrapper transformation can be applied. One important use of rewrite rules in GHC is to implement the deforestation optimization.

[Wadler \[1990\]](#) describes deforestation as a transformation for removing intermediate data structures. For example if we want to map two functions `f` and `g` over a list `xs`, a deforestation transformation would change `map f (map g xs)` into `map (f.g) xs` (where `f.g` is the function composition of `f` and `g`). This transformation removes the temporary list that is allocated for the `map g` operation.

[Gill et al. \[1993\]](#) introduce a technique for deforestation called `foldr/build`. The idea is to write the basic list processing functions (e.g. `map`, `filter`, `concat`) in terms of two functions called `foldr` and `build`. They then use a rewrite rule that transforms combinations of `foldr/build` into a form where no intermediate list is created.

The `foldr/build` transformation is fairly easy to implement, but it cannot get rid of intermediate data structures for all list processing functions. [Coutts et al. \[2007\]](#) introduce *stream fusion* that uses a more powerful technique that works over a wider range of functions including `zip`, `foldl`, and functions working with nested lists. These deforestation algorithms are important examples of transformations that must be done at the high level – there is no hope of performing these optimizations after the code has been lowered to the machine code level. Deforestation is designed to get rid of intermediate data structures, which are an important source of overhead in Haskell. Another source of overhead in Haskell is the calling of statically-unknown functions.

One of the difficulties with implementing functional languages is handling the

application of an unknown function to its arguments. When an unknown function is applied, the compiler cannot simply generate code for a function call. Since we are in a functional language (which can return functions as results), the function could take greater or fewer than the number of arguments to which it is applied. If a function is applied to more arguments than its arity, it should consume as many as it needs and leave the rest for the function it will return as a result. If it is applied to fewer arguments than its arity it should consume those arguments and build partial application as the result (e.g. the result will be a function that will consume the remaining arguments). Either way, the compiler cannot statically generate code for the function call because the function is unknown at compile time.

[Marlow and Peyton Jones \[2006\]](#) evaluate two competing methods for calling unknown functions. The first, called *push/eval*, simply pushes the arguments on the stack and lets the function consume as many arguments as it needs. The second, called *eval/apply*, first evaluates the function to see how many arguments it needs and then calls it with the correct number of arguments. [Marlow and Peyton Jones](#) conclude that the *eval/apply* method is better because it is easier to implement and the performance is slightly better. Unknown function calls are one area where the trace-based optimization techniques used in Htrace would be able improve performance because on the trace, the function is no longer unknown and we could generate a direct call to the function which will be cheaper than using the generic *eval/apply* method. The authors also measured the frequency of unknown calls which they found to be around 20%, so the optimization opportunity does exist.

Another difficulty with implementing a lazy functional language is that values are only created on demand. When executing the program there will be unevaluated values (called *thunks*) interspersed with fully materialized values. The original STG machine used a uniform representation for both *thunks* and normal values. The representation is called a *closure* and consists of a pointer to a piece of code for

evaluating the value along with the free variables needed to perform the computation. To evaluate any closure (also called *scrutinizing* the closure) the STG machine will *enter* the closure by jumping to the code pointed to by the closure. For thunks, the code will evaluate the thunks and then return to the caller. For an already evaluated value, the code will simply return to the caller immediately. For evaluated values, the two indirect jumps are pure overhead.

[Marlow et al. \[2007\]](#) describe a technique called pointer tagging that reduces the overhead when scrutinizing already evaluated closures. They modify the uniform representation of closures so that a closure is now tagged with some extra information in the lower bits of the pointer to the closure. These extra bits indicate whether a value has been evaluated and if so what kind of value it points to (e.g. the data constructor). Tagging the pointers to closures reduces the number of indirect jumps, thereby improving the branch prediction for code that scrutinizes closures.

An interesting entry in the Haskell optimizing literature that can take advantage of pointer tagging is the optimistic evaluation of [Ennals and Peyton Jones \[2003\]](#). They use a combination of static code generation and dynamic profiling to optimistically evaluate lazy expressions (thunks). Statically, they arrange for each expression to either be speculatively evaluated or lazily evaluated based on a dynamic flag for that expression. When the program executes, the Haskell runtime monitors the execution and aborts any speculative evaluation that takes too long. They are trying to exploit the fact that most thunks will be evaluated and so it will be cheaper to evaluate the expression immediately rather than build a thunk and evaluate it later. They achieve an average speedup of 20% over the baseline GHC.

As a final stop on our tour of Haskell optimizations we will look at the work done to optimize the low-level code generated for Haskell programs. These optimizations are performed late in the compilation process after the code has been lowered to an imperative representation of the functional code. Since they work on the low-

level code, they cannot rely on the same semantic information used by the high-level optimizations described earlier. However, these low-level optimizations are important for the overall efficiency of executing Haskell programs and are the focus of this thesis.

The work of [Boquist \[1999\]](#) and [Boquist and Johnsson \[1997\]](#) represent some early attempts to focus on the impact of low-level optimization for Haskell. [Boquist](#) developed the GRIN language for the purpose of performing these optimizations. GRIN is a low-level language that is intended to be used as an intermediate representation when compiling Haskell. One of the main goals of the work was to get rid of overheads from unknown control flow due to application of unknown functions and evaluation of thunks. [Boquist](#) developed a heap analysis that reduces the amount of unknown control flow. The GRIN compiler relies heavily on inlining and requires whole program analysis. While he was able to show good performance for some simple benchmarks, it is not clear that the approach would scale to large programs since his compiler requires access to the whole program at compile time. The approach we take in Htrace is to only include the parts of the program and libraries through which we want to be able to trace. We can manage the overhead by only focusing on the hot parts of the program.

More recently, [Ramsey et al. \[2010\]](#) developed the Hoopl optimization framework for writing low-level optimizations in GHC. Hoopl is based on the idea that combining analysis and transformation can produce better results than iterating analysis passes followed by transformation passes. It uses standard compiler data flow techniques to optimize code. One of the novel contributions of Hoopl is its strongly typed representation of control flow graphs that make it more difficult for the programmer to write invalid program transformations. Although the approach is interesting, they do not actually implement any optimizations for the GHC back end so it is unclear if many opportunities exist for performing these optimizations.

[Terei and Chakravarty \[2010\]](#) present another approach to low-level optimization

of Haskell. Instead of implementing all of the optimizations themselves, they target the LLVM compiler infrastructure which performs many classical optimizations [Lattner and Adve, 2004]. By targeting LLVM and reusing its optimizations they were able to quickly add a large number of optimizations to the GHC back end. For Haskell programs that resemble loops they show that they get a 2-170% improvement. Unfortunately, for typical Haskell codes their results show only an average improvement of 2.4% over the existing native back end that does very little optimization. The lack of effectiveness of the LLVM optimizations for typical Haskell codes was a major motivation for using the more aggressive trace-based optimization techniques presented in this thesis.

Next we survey the major work in runtime optimization. In this thesis, we explored runtime optimization using DynamoRIO and designed Htrace so that it could benefit from the same techniques used by runtime optimizers. We break the presentation into two main pieces: runtime optimization in virtual machines and runtime optimization for native binaries.

7.2 Dynamic Optimization

7.2.1 Virtual-Machine Based Optimizers

Dynamic optimization has been successfully used in virtual machines for object oriented languages for many years. Virtual machines are interpreters for low-level byte-code languages and are an attractive implementation technique because they simplify the porting of languages to new hardware because they provide a stable target across the different hardware platforms. Instead of writing a new code generator for each target platform, we can simply generate byte code for the virtual machine. However, the portability comes at the price of performance because the byte codes are interpreted instead of executed natively on the hardware. To improve performance, compilers can

dynamically generate and optimize machine code for these byte codes. The compilers that operate inside the virtual machines are the focus of this section. We categorize the compilers into two broad categories depending on the basic unit of compilation which can be either a whole method or a program trace.

Method-Based Optimizers

Method-based dynamic compilers use object methods (or functions) as the unit of translation. When a method is selected for dynamic translation to machine code the runtime compiler will generate code and optionally perform optimizations such as inlining and code motion. In this section we survey a variety of method-based dynamic optimizers for Smalltalk, Self, Haskell, and Java.

Dynamic optimization was pioneered by [Deutsch and Schiffman \[1984\]](#) in their implementation of Smalltalk. The key contributions of their design were keeping multiple representations of the program and using an inline method cache. By keeping both the byte code and native code representations simultaneously, they could use the native code when available or simply regenerate it from the byte code when necessary. The inline method cache is an optimization that changes a method lookup to a direct jump to its last destination along with a test to ensure the receiver has not changed. The Smalltalk system showed how to efficiently implement dynamically typed languages in a virtual machine. The implementation of the Self language took this basic design and added several important optimizations.

Self is a dynamically typed object oriented language that uses object prototypes instead of inheritance for code sharing [[Chambers et al., 1991](#)]. An important new optimization implemented for Self is the addition of polymorphic inline caches [[Hölzle et al., 1991](#)]. These caches are similar to the inline cache of Smalltalk except that instead of caching only the last destination for a method call, they cache multiple destinations and perform a type test and direct jump based on the type. These caches

are expanded dynamically as new method targets are found. A side effect of these polymorphic inline caches is that type information is collected at method call sites. [Hölzle and Ungar \[1994\]](#) exploit the type information by dynamically recompiling methods and using the type information to guide the optimizer. The Self language implementation was an important step in the development of dynamic compilers, but it was not until Java became popular that the virtual machine implementation technique really gained a lot of attention.

Java is an object oriented language that supports dynamic code loading which limits its ability to be compiled ahead of time. It has been a rich source of research for virtual machine implementations. Early efforts include the compiler of [Adl-Tabatabai et al. \[1998\]](#), which focused on fast code generation by directly generating machine code from the Java byte code instead of building a separate intermediate representation as is common for static compilers of other languages. [Burke et al. \[1999\]](#) built the Jalapeno Java compiler that focuses on adaptive compilation, similar to the Self system of [Hölzle and Ungar \[1994\]](#). Jalapeno includes a fast code generator for the first-time compilation along with an optimizing compiler that is invoked to recompile frequently executed methods. Java virtual machines remain an active area of research with recent dynamic compilers implementing a variety of advanced analyses and optimizations including escape analysis, object inlining, and array bounds check elimination [[Kotzmann et al., 2008](#)]. Although Java is among the most successful languages implemented on a virtual machines, the technique has been used on a variety of other languages including Haskell.

[Wakeling \[1998\]](#) describe a virtual machine implementation for running Haskell on embedded systems. Unlike our work, his work focuses on reducing code size rather than on improving performance. They show that by dynamically compiling byte-code instructions they can reduce the code size by 67% while only slowing down the code by 25%.

All of the virtual machines described in this section compile byte codes at the resolution of a method (or a function in the case of Haskell). In the next section we discuss virtual machine implementations that work on program traces as the basic unit of compilation.

Trace-Based Optimizers

Trace-based compilers change the unit of optimization from a function or method to a sequence of instructions called a *trace*. The motivation behind compiling instruction traces instead of functions is to expose the new opportunities that appear based on the actual control flow in the program. Early work on optimizing traces include the trace scheduling algorithm of Fisher [1981] for instruction scheduling. Hank et al. [1995] were early advocates of changing the compilation focus from the programmer defined functions to compiler designated regions. These two works are exemplar of much of the early work on trace-based compilation that focus on building traces in a static compiler. In this section we will limit the discussion of trace-based compilers to the dynamic compilers found in virtual machines.

Suganuma et al. [2006] describe a dynamic region-based compiler for Java. Their regions are slightly different from traces because the regions are not necessarily a linear sequence of instructions. A region may still contain internal control flow, but they are constructed using runtime profiling data so that rarely executed code is not included in the region. Using the region based approach, they were able to achieve an average performance improvement of 4% and a reduction in compilation time of 10-30%. The region-based approach to compilation is similar to trace-based compilation, but the trace-based approach builds traces that contain only linear control flow. The remainder of this section surveys trace-based compilers in virtual machines.

Gal [2006] describes a trace-based Java compiler for resource constrained devices such as cell phones. His compiler focuses on efficient runtime code generation and was

able to generate code much faster than existing compilers while producing code that was competitive with heavyweight desktop virtual machines. A major contribution of his work was the introduction of trace trees [Gal and Franz, 2006] for building and optimizing program traces. The trace tree is initially constructed using the Next Executing Tail (NET) heuristic of Dynamo [Duesterwald and Bala, 2000] and then *extended* by keeping track of frequent side exits from the trace. If a side exit is taken frequently, the trace tree is rebuilt to include a path for the (former) side exit. The advantage of trace trees over NET traces is that trace trees build longer traces because the trace prefixes are shared among many traces. Sharing the trace prefixes allows him to build traces that flow through nested loops, unlike NET which would build a separate trace for each loop. We could extend Htrace to support trace trees by using shadow tracing to find the hot side exits. Gal’s work focused on trace-based compilation as a way to overcome limitations of embedded architectures, but more recent work embraces trace-based compilation as a way to improve performance on a variety of architectures.

Recent virtual machines implementations have adopted trace-based compilers for several languages including compilers for JavaScript, Microsoft’s Common Intermediate Language (CIL), and Java. Gal et al. [2009] describe the TraceMonkey JavaScript compiler. In this work, they describe an improved algorithm for generating trace trees for nested loops and show how to generate type-specialized code for these traces. Using their compiler they can achieve speedups of up to 20x. Bebenita et al. [2010b] describe the SPUR trace compiler for CIL. They show that targeting JavaScript to the CIL and then trace compiling the CIL is just as effective as directly trace compiling the JavaScript. They also describe several optimizations for traces including a store-load propagation that uses alias analysis to avoid writes to the heap inside the trace. Dalvik is a virtual machine with a trace-based compiler for Java [Cheng and Buzbee, 2010] used for running applications on Android cell phones. The trace

compiler is in the early stages, but initial results show a performance gain of up to 5x. These works show the continued interest in improving the performance of trace-based compilers. The final works we survey describe efforts to adapt trace compilers to work with existing systems.

[Inoue et al. \[2011\]](#) describe their effort to add a trace compiler to the existing method-based compiler in the IBM production JVM. The work is particularly interesting because it allows a direct comparison of a trace-based compiler with a mature method-based compiler. They found that the trace compiler was able to generate better code than the method compiler, but that it incurred larger overhead from recording and monitoring traces. The overall result was that they saw the performance range from 21.5% slower to 26.4% faster compared to the method-based compiler. They conclude that trace-based compilation is viable, and it probably is best paired with a traditional method-based compiler to achieve the best of both techniques. Like the IBM compiler, many of the compilers we have seen so far are able to achieve a great performance improvement over the interpreted code because they eliminate the overhead of interpretation. In the next work, the trace compiler is added to a system that uses no interpreter at all making it more difficult for them to get the easy performance boost.

[Bebenita et al. \[2010a\]](#) describes the Maxpath Java compiler. Maxpath is a trace-based compiler that builds and optimizes program traces without the help of an interpreter for introspection of the executing code. They identify two main difficulties with using a trace-based compiler without an interpreter. First, building the traces is more difficult because they do not have an interpreter that can introspect the code as it executes and record the traces. Second, because they are using a system that performs compilation instead of interpretation as a baseline, the trace compiler must be much more aggressive to get a performance improvement. Their results show that for small kernels the technique did well, but did not work as well on large programs.

They speculate it is because the large programs do not have loop kernels that are amenable to their trace optimization. Their problems are similar to the difficulties we encountered with improving performance and building traces on many of the Hackage benchmarks.

In this section we looked at optimizing compilers that work in the context of a virtual machine. These compilers are the most common type of dynamic optimizers and have an advantage over binary optimizers because they can take advantage of the high-level semantic information from the byte code. In the next section we review compilers that dynamically optimize programs directly on the binary level.

7.2.2 Dynamic Binary Optimizers

Dynamic binary optimization is a technique for optimizing a program by rewriting the machine code at runtime. It is a relatively new technique that has yet to see adoption in production compilers. In this section we discuss the major advances in binary optimization and some of the common issues involved.

The Dynamo compiler of [Bala et al. \[2000\]](#) is the genesis of dynamic binary optimization systems. Dynamo works by interpreting the machine code of a program and building traces of the frequently executed portions of the program. The traces are linked together in a *fragment cache* where they can be directly executed as machine code instead of interpreted by software. Several optimizations are performed on the traces including redundancy removal, copy propagation, constant propagation, and code motion. Dynamo also included a ‘bail out’ option that will stop all optimization attempts and fall back to native execution. They report results that show an average speedup of 9% over heavily optimized binaries on the SPECInt95 benchmarks. The performance improvement comes largely from the indirect branch removal and code layout that occurs when building traces. Dynamo showed a novel way of optimizing programs that was expanded on by the DynamoRIO system.

[Bruening \[2004\]](#) describes the DynamoRIO system that builds on the ideas found in Dynamo. As with Dynamo, DynamoRIO is a dynamic binary optimizer that builds program traces at runtime. Its main contribution over Dynamo is to provide a general framework for optimizing a program as it executes. DynamoRIO works with modern programs that use multiple threads, dynamic linking and even self modifying code. In contrast with Dyanamo, the performance results for the SPEC benchmark suite show that DynamoRIO does not perform well on the SPECInt benchmarks, but does get a 12% average improvement on the SPECfp benchmarks. The reason for this difference is likely the variance in the cost of basic operations (e.g. indirect branches) on the hardware platforms used for the two experiments: Dynamo was run on a PA RISC chip, but DynamoRIO was run on an x86 processor. DynamoRIO was used for some of the work in this thesis, but there are several other binary optimization systems worth mentioning.

Adore is a trace-based optimizer that uses hardware performance counters to selectively optimize a running program [[Lu et al., 2004](#), [2003](#)]. Adore periodically samples a program to record the most recent branches taken. Once a path has been sampled enough times it will be used as the starting point of a trace. It also includes a mechanism to detect phase changes in a program that is used as a signal to throw out the old and build new traces. Once a trace is built they perform a few optimizations, the most important one being data cache prefetching. To insert the prefetches they use performance counters to track the locations of the loads that cause L2 or L3 cache misses and insert prefetches for the loads that cause the greatest percentage of overall latency. Their results show that they can achieve a performance improvement from 3% to 107%, while keeping the minimum performance degradation to -3%. Their overheads are generally lower than DynamoRIO because they limit the interference with the original application by using sampling and hardware performance counters to build the traces. However, they only show results for Itanium and we have see with

Dynamo and DynamoRIO that the underlying architecture can have a big impact on the performance results of binary optimizers. To further verify the usefulness of binary optimization, Adore was used to optimize the BLAST application.

[Das et al. \[2005\]](#) describe their experience optimizing the Basic Local Alignment Search Tool (BLAST) application using the Adore dynamic optimizing compiler. They found that the static compilers were degrading performance by aggressively inserting data prefetches that were interfering with the real data loads. They were able to improve the performance of BLAST by up to 58% by only inserting data prefetches where they were needed which was determined by monitoring hardware performance counters during runtime. Their results show that dynamic optimization has a benefit over static optimization because the compiler can make decisions based on how the program data interacts with the actual hardware. The next example shows how the dynamic prefetching optimization can be applied to improve the performance of parallel programs.

[Kim et al. \[2007\]](#) describe the COBRA system for dynamic binary optimization of multi-threaded programs. Their system is similar to Adore because they use hardware events to selectively optimize a program at runtime. They focus on two optimizations: providing the correct prefetch hints for cache coherent memory accesses and reducing the aggressiveness of prefetching to reduce memory bus contention. Their results show that on an Itanium system they can achieve a speedup of up to 15% on the NAS parallel benchmarks with an average speedup of 4.7%.

Now that we have introduced the major binary optimization systems, we will take a look at a few works that focus specifically on techniques to improving the quality of traces used by these systems.

Selecting traces is an important component in dynamic binary optimizers because the quality of the traces ultimately determine the performance of program. [Hiniker et al. \[2005\]](#) describe a new algorithm for building traces that improves on the Next

Executing Tail (NET) algorithm used in Dynamo [Duesterwald and Bala, 2000]. They identify two problems with existing trace generation algorithms: trace separation and code duplication. Trace separation is the problem that occurs when related code paths in the original application are placed far apart in the code cache. Code duplication is the problem that occurs when code is replicated across many traces. To combat these two problems the authors present a new algorithm for building traces called Last-Executed Iteration (LEI) and an algorithm for combining traces. The LEI algorithm uses a circular buffer to keep track of the branches taken in the machine code interpreter. When it detects that a loop has occurred a certain number of times, it will build a trace for the iteration that just occurred. Using a simulator they find that their algorithm builds traces that contain more full cycles and they require fewer traces to cover 90% of the instructions executed.

Zhao et al. [2008] propose that traces can be improved by increasing their length and present a technique for lengthening traces without increasing early exits from the trace. They classify traces by the most frequent way that the trace is exited. If the trace exits to another hot trace it is called a hot-trace exit and if the trace jumps back to itself it is called a self exit. They lengthen the self-exit traces by unrolling the trace and they lengthen the hot-trace-exit traces by creating a new trace that is the concatenation the two traces. They find that lengthening traces increase the number of opportunities for local value numbering optimization by between 16-23% depending on how aggressively they unroll the traces. These results show the importance of longer traces for finding optimization opportunities.

All of the systems described in this section perform the trace building and optimization at runtime. After an initial attempt at runtime optimization, we decided that building traces at runtime was too expensive for Haskell programs. In Htrace, we use a separate profiling run to find the traces and then build and optimize the traces offline. Using profiling data to optimize a program is called feedback directed

optimization, and we survey the important works from that area in the next section.

7.3 Feedback Directed Optimization

Feedback directed optimization (FDO) is an optimization technique that uses profiling data to improve the effectiveness of compiler transformations. In this section we survey techniques for collecting profile data – with an emphasis on path profiles – and related work on how to optimize a program based on profile data.

7.3.1 Collecting Profile Data

In this section we survey the major works on how to collect profiling data. There are two primary techniques for collecting profiles: instrumentation and sampling based profiling. The instrumentation-based approach has the advantage that the profiles are exact because the instrumentation is inserted at each desired sample point. The sampling-based approach periodically measures the program to collect the profiling data. The advantage of a sampling-based profiler is that it can control the overhead it introduces by modulating the sampling frequency. Another difference between the approaches is that instrumentation-based profiling has been used to collect execution path profiles, while the sampling based approach has frequently been used for collecting call path data without the details of the execution path inside the function. Our focus is on instrumentation-based path profilers, but we will briefly discuss the sampling-based approach as well.

The seminal algorithm for recording path profiles is the work of [Ball and Larus \[1996\]](#). In this paper, they show how to efficiently collect path profiles for a single procedure. Their key insight is to represent paths by unique integers and insert instrumentation so that as a path executes it increments a value stored in a register. The increments are arranged so that every acyclic path through the procedure will

leave a different final value in the register. They have a transformation that will take a cyclic graph and transform it to eliminate the cycles for profiling purposes. At the end of each path they increment a counter for that path by using the index value stored in the register. The path counters are either stored in an array or a hash table depending on the number of paths in the graph. They optimally arrange the register increments by using a technique from their earlier work that inserts instrumentation on a subset of the edges in the procedure based on a spanning tree of the control flow graph [Ball and Larus, 1994]. They report an average overhead of 30.9% on 18 of the SPEC benchmarks, compared to an overhead of 16.1% for edge profiling instrumentation. The Ball-Larus algorithm is the basis of many path profiling systems. We review several of the major works that build off of this algorithm.

Targeted Path Profiling (TPP) is an extension by Joshi et al. [2004] to the Ball and Larus algorithm described above. They developed the algorithm with the idea that it would be used in a dynamic compiler and assume that they already have some initial edge profiling data that was collected before inserting the path profiling instrumentation. The edge profile data allows them to reduce the number of paths they will profile by omitting counters updates for the cold edges. Reducing the number of paths is a big win for functions with many paths because the counter updates change from an expensive hash-table lookup to a much more efficient array indexing operation. Compared to the standard Ball-Larus algorithm, they report a reduction in overhead of one-half on the SPEC95 benchmarks and one-third on the SPEC2000 benchmarks.

The Practical Path Profiling (PPP) algorithm Bond and McKinley [2005] is an extension of TPP by that further reduces the overhead of the original Ball-Larus algorithm. Similar to TPP, the PPP algorithm uses path profile data to reduce the number of paths they must instrument. They further reduce overhead by inserting less instrumentation on the hot paths in the function. As a final contribution, they

define a new metric for measuring the accuracy and coverage of their path profiling algorithm.

All of the algorithms presented so far operate on a single function. They must build the control flow graph for the function in order to correctly insert instrumentation. While it would be interesting to try and apply these algorithms to Htrace, the major difficulty is that Htrace must build paths across multiple functions. Then next references discuss techniques for building profile paths across procedure boundaries.

Melski [2002] extends the Ball-Larus algorithm to handle inter-procedural program paths. He calls his algorithm the “functional” approach to path profiling. The “functional” name is used because the counter updates that are inserted on the edges to keep track of the path number are linear functions instead of simple increments. To label all the paths in the program he must build the program *supergraph*, which is a collection of procedure control-flow graphs that have been connected according to the call graph. He reports that his algorithm finds 2508 paths on average across six of the SPECint95 benchmarks, although for most benchmarks 50 paths are sufficient to cover 80% of the programs execution time. The profiling overhead for his technique is 235% on average. The main drawback of his approach from the perspective of Htrace, is that he needs an inter-procedural control-flow graph in order to label paths in the program. In addition to the memory overhead, the supergraph is difficult to build precisely in the presence of indirect calls, which are very common in Haskell programs.

Larus [1999] introduces an algorithm for finding Whole Program Paths (WPP). A WPP is a description of the entire control flow of a program. To make the description usable, Larus compresses the program trace with the SEQUITUR compression algorithm, which uses a grammar to generate the textual representation of the trace. The result is a representation of the program trace as a DAG, where interior nodes represent repeated sequences and leaf nodes represent the actual acyclic paths in the

program. The acyclic paths in the leaf nodes are numbered according to the same scheme given in the original path profiling algorithm of [Ball and Larus \[1996\]](#). He presents an algorithm for finding hot paths in the WPP. These paths can span across multiple functions and would be the targets of path-based compiler optimizations. Unlike the supergraph used by Melski, the WPP does not require complete inter-procedural control-flow knowledge so the WPP could possibly be used as a trace finding technique in Htrace.

All of the work we have seen so far are based off of the Ball-Larus algorithm to some extent. The next work is a break with that tradition, and it is important because it was the basis of the trace finding algorithm for Htrace.

[Duesterwald and Bala \[2000\]](#) proposed the Next Executing Tail (NET) scheme for finding hot paths in a program. This scheme was used by Dynamo² and DynamoRIO for finding hot paths. It is also the basis used in this thesis for finding hot traces with Htrace. The NET algorithm finds paths by only using counters for *trace header* blocks, which are the targets of backward branches. Once the header becomes hot, the next execution leaving that header is recorded as the trace. Their primary motivation was to find an inexpensive profiling technique that could be used in a dynamic optimizer. They experimented with a variety of hotness thresholds (which they call prediction delay) and found that a small threshold of 50 gave the best performance. With the small hotness thresholds used in a dynamic compiler, they found that NET gave comparable coverage of hot traces to the more expensive Ball-Larus algorithm. They were less clear about the benefits of NET compared to Ball-Larus when using larger thresholds, but their results seem to suggest the algorithm produces similar hot-trace coverage. We used a variant of the NET algorithm in this thesis because it works well for inter-procedural traces and is easy to implement. It is possible that a more heavyweight algorithm based on Ball-Larus would allow Htrace

²It was called MRET in the original Dyanamo paper [[Bala et al., 2000](#)]

to produce better traces, but we leave that for future work.

[Ammons et al. \[1997\]](#) describe how to extend the basic Ball-Larus algorithm to account for calling context. They introduce the *calling context tree* (CCT) which is a data structure that represents the context in which a procedure was called. The CCT is used to collect separate intra-procedural path profiles for each unique calling context. They argue that the calling context may reveal important differences in the hot paths, and that these differences will be blurred when path profiles are aggregated over all calling contexts. Although they call the CCT a tree, they do allow back edges in the tree to handle recursive calls. They report an average overhead of 70% when combining the CCT with path profiling. This paper is important because it introduces the idea of the CCT, which allows a limited form of inter-procedural paths because the per-context path profiles can be combined with the CCT to build a path through multiple procedures. The calling context tree has become an important tool for many profiles, but collecting the CCT using instrumentation may have too high an overhead for some applications.

[Froyd et al. \[2005\]](#) describe a sampling based approach to finding the calling context tree. They built a tool called *csprof* to sample the program at repeated intervals and walk the call stack to build the CCT. To make the stack walking efficient, they cache the current state of the stack at each sample point. Then, at the next sample point they only have to walk up the part of the stack that has changed since the last sample. They are able to collect profiling data for unmodified and fully optimized binary programs. Because they work with binaries, they are able to collect call path profiles through libraries. It would be interesting to try and apply the sampling based approach of collecting profile data to Htrace. One problem is that it may be difficult to turn a CCT into a realized trace, because the CCT is a summary of the observed behavior rather than an actual path through the code. Another issue is the frequent use of tail calls and recursion in Haskell could make it difficult for the

sampling-based approach to correctly elucidate the entire CCT. Exploring the use of sampling-based profiles to build traces with Htrace is an interesting area of future work.

7.3.2 Optimizing with Profile Data

One of the first entries in the feedback directed optimization literature is the work of [Chang et al. \[1991\]](#), who describe an optimizer that uses profile information to assist with classic compiler optimizations. They first profile a program to collect basic block and edge counts. The counts are then used to build program traces that they call *super blocks*, which are linear sequences of blocks that can only be entered from the top block. The super blocks are formed based on edge counts, which was shown by [Ball and Larus \[1996\]](#) to be inferior to path profiles. The inferiority of edge profile counts to path counts is why we use path profiles for Htrace. After giving an algorithm to build super blocks, they then describe several optimizations that operate over the increase scope, such as redundant store elimination and loop invariant code removal. They report an average speedup of 15% on 13 benchmarks with a maximum speedup of 42%. The Htrace system we built could take advantage of this work by forming super blocks from the traces we find and implementing the super-block optimizations described in this paper.

[Pettis and Hansen \[1990\]](#) describe a code placement algorithm that uses profile data to place frequently executed instructions close together. They investigated code layout at two different scopes: procedure and basic block. The procedure layout algorithm uses the linker to place functions that frequently call each other close together in the code. The basic block layout algorithm uses edge profiling data to build long chains of basic blocks whose common path uses the fall-through case on the branch instructions. The layout improves instruction cache usage and reduces the number of taken branches. They also describe a technique called *procedure splitting*

that moves blocks that were not executed in the profiling run to a far away location in the code so that the frequently executed code across all procedures is located close together. The layout algorithms described in this work could definitely be applied as optimizations in Htrace after all the functions on the trace have been inlined to a single procedure.

[Arnold et al. \[2000\]](#) describe a study of inlining heuristics that compares a static inlining strategy to ones that makes use of profiling data. The static heuristic attempts to inline the methods with the least cost, where the cost is measured in code size. The two dynamic heuristics use either node weights or edge weights collected from a profiling run to weight the benefit of inlining a particular function. They found that using edge weights produces an inlining heuristic that has a much greater benefit than both the static and node-weighted heuristics. Additionally, the edge weights allow for inlining a larger number of dynamic calls with a smaller increase in code size. Our Htrace system essentially uses node counts to find the hot functions, and uses one pass to find the functions to inline starting from the trace header. Using a global notion of edge counts as in this work could possibly allow for better trace formation, but more study is needed because the most frequently weighted edges will not necessarily correspond to an actual execution trace.

[Li et al. \[2010\]](#) implement a system called LIPO, which stands for lightweight inter-procedural optimization. Their main contribution is a lightweight integration of feedback directed optimization with inter-procedural optimization. LIPO differs from existing systems, of which Htrace is one example, where the inter-procedural optimization is performed by aggregating different modules into a large single module and feeding that module to the compiler. Instead, LIPO runs an inter-procedural analysis (IPA) to the end of the profiling run and includes the analysis result with the profiling data. The analysis builds the weighted call graph based on the profiling run. The program source modules will be grouped into clusters based on their affinity

in the call graph. When the compiler goes to optimize the program based on the FDO data, it also uses the IPA data to read in the other source modules and apply existing inter-module optimization on the increased scope. Their results show that they achieve an average improvement of 2.4-4.4% over standard FDO, and their time and space overhead is much reduced compared to existing systems.

Many of the works we have surveyed so far use profiling information based on basic block or edge counts. Several works have focused on profiling actual paths in an application and performing optimizations on those paths. [Young \[1998\]](#) describes an approach to *path based compilation* that profiles a program to find the paths and then optimizes along those paths. He gives an algorithm for profiling paths that is asymptotically as efficient as edge and node profilers. He then shows how to use path profiles in two optimizations. The first optimization seeks to improve branch prediction by cloning separate paths through a group of conditional branches. The second optimization is a instruction scheduling algorithm that creates super blocks based on the path profiles and schedules the instructions in the super block.

Two other examples of path-based optimization are the works of [Gupta et al. \[1997\]](#) and [Gupta et al. \[1998\]](#). They describe two different algorithms that take advantage of path-profile data. The first optimization is partial dead code elimination using prediction. They find expressions that are partially dead (meaning they is killed on some, but not all paths) and where possible move the dead computations off the hot paths. Their second optimization is a partial redundancy elimination algorithm (PRE) that uses path profiles to decide where to speculatively evaluate expressions. The speculative evaluations are placed in the cold path at points that will make the expression fully redundant along the hot path. The fully redundant expression can then be removed in the hot path. These algorithms are good examples of optimizations we could implement for Htrace that would enhance the benefits we get from low-level optimizations after we have built the program traces.

Chapter 8

Conclusion

This thesis focuses on improving the effectiveness of low-level compiler optimizations for Haskell. We claim three major contributions. First, we examine the low-level behavior of Haskell programs from the Fibon benchmark suite and compare them to C, C++, and Fortran programs from the SPEC benchmark suite. Our results showed that the Haskell programs contain many more indirect jumps than the SPEC programs which gave us the idea to use trace-based optimization techniques to improve the opportunities for the low-level optimizer. Our second contribution was to explore using a runtime binary trace-based optimizer on Haskell programs. We discovered that the trace-based optimizer was able to find many traces in Haskell programs, but the traces were of poor quality and building and maintaining the traces at runtime induced a tremendous amount of overhead. The large runtime overhead led us to develop a static trace-based optimizer that uses profiling runs to find the hot program traces. The static optimizer restructures the code around the trace and performs traditional compiler optimizations over the increased scope. Our final results show that we can improve the effectiveness of low-level optimizations by up to 86%, with an average speedup of 5% across 32 benchmarks.

We became interested in the applicability of low-level optimizations to Haskell

code when we noticed that many of the Fibon benchmarks did not benefit from the LLVM backend. To understand why the low-level optimizations did not help we attempted to quantify the difference between Haskell programs and C/C++/Fortran programs in terms of the low-level code. Our measurements found that while the Fibon benchmarks are similar to the SPECint benchmarks in terms of instruction-type mix and basic block length, they tend have many more indirect jumps. We believe it is these indirect jumps that are partially responsible for the general failure of low-level optimizations to improve the performance of Haskell programs. We proposed using trace-based optimization techniques to improve the effectiveness of low-level optimizations for Haskell by increasing the scope available to the optimizer.

Our first attempt at building program traces for Haskell used the DynamoRIO runtime binary optimizer. DynamoRIO was able to find many traces in the Haskell programs, but it caused a large hit in performance. The main cause of the overhead was the hash table lookups that occurred at many of the indirect branches. We saw that Haskell has many indirect branches, so these table lookups can be very expensive. Additionally, the traces found by DynamoRIO include many paths through the GHC runtime functions, such as the garbage collector, and there is no easy way to only build traces that specifically target only the unknown control flow that come from the way Haskell source code is lowered to machine code. Our observations with running Haskell programs through DynamoRIO led us to design our own trace-based optimization system that works to overcome the low-level inefficiencies of GHC's execution model.

We built and evaluated Htrace, a system for restructuring low-level Haskell code around program traces. The implementation of Htrace allowed us to experiment with the effectiveness of trace-based optimization for Haskell. The design of the system allows for a variety of trace-finding and trace-building heuristics. Even with the simple set of heuristics we selected for our experiments we found that we see consid-

erable performance benefits. The primary improvement we make is to increase the scope available to the low-level optimizer by restructuring the program around common execution paths. The performance benefit is greatest when we reveal hot loops through traces that often run to completion. Programs that spread their execution across many traces or for which the trace building heuristics fail will not see any benefit from our technique. We also have a number of programs where we do not see a great benefit because of a failure in the low-level optimizer to improve the code. These programs have potential for a performance increase with an improvement to the trace-building heuristics and the low-level optimizer.

The work in this thesis builds a foundation for future exploration of low-level optimization for Haskell code. Future avenues for further research include refining the trace-finding heuristics and exploring low-level optimizations tailored to the code contained in the Haskell traces. Finding the correct program traces is important for reducing overhead and increasing the effectiveness of the optimizations. We presented a simple scheme for finding traces and found that it is effective for a good number of programs, but there is still much room for improvement. For low-level optimization, we simply used the existing optimizations found in LLVM. Now that we have shown we can effectively increase the optimization scope by building program traces, it would be worthwhile to explore optimizations that are profitable for the code patterns seen in Haskell programs. In particular, we should develop optimizations that target redundant loads and stores from the separately managed call stack.

Although we have focused exclusively on Haskell in this thesis, we can place the work in a broader context. Our trace-based optimization technique was predicated on the notion that the natural control flow was obscured by indirect jumps. Because we operate on language agnostic low-level code, our technique could work with any code that exhibits the same code shape as Haskell. The effectiveness of this technique will depend on the degree to which the obscured control flow is unraveled by the program

trace. The applicability of our technique will vary according to how the high-level language features are manifested in the low-level code. The defining features of Haskell that cause the unknown control flow are higher order functions, lazy evaluation, and a separately managed call stack. A language with some combination of these features is certainly a candidate for improvement by the technique presented in this thesis.

We have made a case for trace-based optimization of Haskell programs. The main insight that made this work is that we can only optimize what we can see. Program traces allow us to see more of the code and increases the scope over which we can apply our optimizations. In many ways optimizing low-level Haskell code is no different than optimizing traditional languages. We look for the part of the program that are executed frequently, such as a loop, and reduce the redundant computation in that area of the code. Optimizing loops is just as important for Haskell as it is for traditional languages. The main difference for Haskell is that the loops can be quite hidden at compile time. Our goal is to bring the low-level loops back to Haskell.

Bibliography

- A.-R. Adl-Tabatabai, M. Cierniak, G.-Y. Lueh, V. M. Parikh, and J. M. Stichnoth. Fast, effective code generation in a just-in-time java compiler. In *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, PLDI '98, pages 280–290, New York, NY, USA, 1998. ACM. ISBN 0-89791-987-4. doi: <http://doi.acm.org/10.1145/277650.277740>. URL <http://doi.acm.org/10.1145/277650.277740>.
- G. Ammons, T. Ball, and J. 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*, PLDI '97, pages 85–96, New York, NY, USA, 1997. ACM. ISBN 0-89791-907-6. doi: 10.1145/258915.258924. URL <http://doi.acm.org/10.1145/258915.258924>.
- M. Arnold, S. Fink, V. Sarkar, and P. F. Sweeney. A comparative study of static and profile-based heuristics for inlining. In *Proceedings of the ACM SIGPLAN workshop on Dynamic and adaptive compilation and optimization*, DYNAMO '00, pages 52–64, New York, NY, USA, 2000. ACM. ISBN 1-58113-241-7. doi: 10.1145/351397.351416. URL <http://doi.acm.org/10.1145/351397.351416>.
- V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, PLDI '00, pages 1–12, New York, NY, USA, 2000. ACM. ISBN 1-58113-199-2. doi: <http://doi.acm.org/10.1145/349299.349303>. URL <http://doi.acm.org/10.1145/349299.349303>.
- T. Ball and J. R. Larus. Optimally profiling and tracing programs. *ACM Trans. Program. Lang. Syst.*, 16(4):1319–1360, 1994. ISSN 0164-0925. doi: 10.1145/183432.183527. URL <http://doi.acm.org/10.1145/183432.183527>.
- T. Ball and J. R. Larus. Efficient path profiling. In *Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture*, MICRO 29, pages 46–57, Washington, DC, USA, 1996. IEEE Computer Society. ISBN 0-8186-7641-8. URL <http://dl.acm.org/citation.cfm?id=243846.243857>.
- M. Bebenita, F. Brandner, M. Fahndrich, F. Logozzo, W. Schulte, N. Tillmann, and H. Venter. Spur: a trace-based jit compiler for cil. In *Proceedings of the ACM international conference on Object oriented programming systems languages and*

- applications*, OOPSLA '10, pages 708–725, New York, NY, USA, 2010a. ACM. ISBN 978-1-4503-0203-6. doi: <http://doi.acm.org/10.1145/1869459.1869517>. URL <http://doi.acm.org/10.1145/1869459.1869517>.
- M. Bebenita, M. Chang, G. Wagner, A. Gal, C. Wimmer, and M. Franz. Trace-based compilation in execution environments without interpreters. In *Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java*, PPPJ '10, pages 59–68, New York, NY, USA, 2010b. ACM. ISBN 978-1-4503-0269-2. doi: <http://doi.acm.org/10.1145/1852761.1852771>. URL <http://doi.acm.org/10.1145/1852761.1852771>.
- M. D. Bond and K. S. McKinley. Practical path profiling for dynamic optimizers. In *Proceedings of the international symposium on Code generation and optimization*, CGO '05, pages 205–216, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2298-X. doi: <http://dx.doi.org/10.1109/CGO.2005.28>. URL <http://dx.doi.org/10.1109/CGO.2005.28>.
- U. Boquist. *Code Optimisation Techniques for Lazy Functional Languages*. PhD thesis, Chalmers University of Technology, 1999.
- U. Boquist and T. Johnsson. The grin project: A highly optimising back end for lazy functional languages. In *IFL '96: Selected Papers from the 8th International Workshop on Implementation of Functional Languages*, pages 58–84, London, UK, 1997. Springer-Verlag. ISBN 3-540-63237-9.
- D. Bruening. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. PhD thesis, MIT, 2004.
- D. Bruening, T. Garnett, and S. Amarasinghe. An infrastructure for adaptive dynamic optimization. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, CGO '03, pages 265–275, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-1913-X. URL <http://portal.acm.org/citation.cfm?id=776261.776290>.
- M. G. Burke, J.-D. Choi, S. Fink, D. Grove, M. Hind, V. Sarkar, M. J. Serrano, V. C. Sreedhar, H. Srinivasan, and J. Whaley. The jalapeno dynamic optimizing compiler for java. In *Proceedings of the ACM 1999 conference on Java Grande*, JAVA '99, pages 129–141, New York, NY, USA, 1999. ACM. ISBN 1-58113-161-5. doi: <http://doi.acm.org/10.1145/304065.304113>. URL <http://doi.acm.org/10.1145/304065.304113>.
- M. M. T. Chakravarty, R. Leshchinskiy, S. Peyton Jones, G. Keller, and S. Marlow. Data parallel haskell: a status report. In *DAMP '07: Proceedings of the 2007 workshop on Declarative aspects of multicore programming*, pages 10–18, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-690-5. doi: <http://doi.acm.org/10.1145/1248648.1248652>. URL http://portal.acm.org/ft_gateway.cfm?id=1248652&type=pdf&coll=GUIDE&dl=GUIDE&CFID=109512594&CFTOKEN=13068509.

- C. Chambers, D. Ungar, and E. Lee. An efficient implementation of self, a dynamically-typed object-oriented language based on prototypes. *Lisp Symb. Comput.*, 4:243–281, 1991. ISSN 0892-4635. doi: 10.1007/BF01806108. URL <http://portal.acm.org/citation.cfm?id=113819.113823>.
- P. P. Chang, S. A. Mahlke, and W. mei W. Hwu. Using profile information to assist classic code optimizations. *Softw. Pract. Exper.*, 21(12):1301–1321, 1991. ISSN 0038-0644. doi: <http://dx.doi.org/10.1002/spe.4380211204>.
- B. Cheng and B. Buzbee. A jit compiler for android’s dalvik vm. In *Google IO*, 2010.
- K. D. Cooper and L. Torczon. *Engineering a Compiler*. Morgan Kaufmann Publishers Inc., 2nd edition, 2012.
- K. D. Cooper, L. T. Simpson, and C. A. Vick. Operator strength reduction. *ACM Trans. Program. Lang. Syst.*, 23:603–625, 2001. ISSN 0164-0925. doi: <http://doi.acm.org/10.1145/504709.504710>. URL <http://doi.acm.org/10.1145/504709.504710>.
- D. Coutts, R. Leshchinskiy, and D. Stewart. Stream fusion: from lists to streams to nothing at all. In *ICFP ’07: Proceedings of the 12th ACM SIGPLAN international conference on Functional programming*, pages 315–326, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-815-2. doi: <http://doi.acm.org/10.1145/1291151.1291199>. URL http://portal.acm.org/ft_gateway.cfm?id=1291199&type=pdf&coll=GUIDE&dl=GUIDE&CFID=76302416&CFTOKEN=52389390.
- A. Das, J. Lu, H. Chen, J. Kim, P.-C. Yew, W.-C. Hsu, and D.-Y. Chen. Performance of runtime optimization on blast. In *Proceedings of the international symposium on Code generation and optimization*, CGO ’05, pages 86–96, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2298-X. doi: <http://dx.doi.org/10.1109/CGO.2005.25>. URL <http://dx.doi.org/10.1109/CGO.2005.25>.
- L. P. Deutsch and A. M. Schiffman. Efficient implementation of the smalltalk-80 system. In *Proceedings of the 11th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL ’84, pages 297–302, New York, NY, USA, 1984. ACM. ISBN 0-89791-125-3. doi: <http://doi.acm.org/10.1145/800017.800542>. URL <http://doi.acm.org/10.1145/800017.800542>.
- E. Duesterwald and V. Bala. Software profiling for hot path prediction: less is more. In *Proceedings of the ninth international conference on Architectural support for programming languages and operating systems*, ASPLOS-IX, pages 202–211, New York, NY, USA, 2000. ACM. ISBN 1-58113-317-0. doi: <http://doi.acm.org/10.1145/378993.379241>. URL <http://doi.acm.org/10.1145/378993.379241>.
- R. Ennals and S. Peyton Jones. Optimistic evaluation: an adaptive evaluation strategy for non-strict programs. In *Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*, ICFP ’03, pages 287–298, New

- York, NY, USA, 2003. ACM. ISBN 1-58113-756-7. doi: <http://doi.acm.org/10.1145/944705.944731>. URL <http://doi.acm.org/10.1145/944705.944731>.
- Fibon. Fibon. <https://github.com/dmpots/fibon>, 2010.
- J. Fisher. Trace scheduling: A technique for global microcode compaction. *IEEE Transactions on Computers*, 30:478–490, 1981. ISSN 0018-9340. doi: <http://doi.ieeecomputersociety.org/10.1109/TC.1981.1675827>.
- N. Froyd, J. Mellor-Crummey, and R. Fowler. Low-overhead call path profiling of unmodified, optimized code. In *Proceedings of the 19th annual international conference on Supercomputing*, ICS '05, pages 81–90, New York, NY, USA, 2005. ACM. ISBN 1-59593-167-8. doi: [10.1145/1088149.1088161](http://doi.acm.org/10.1145/1088149.1088161). URL <http://doi.acm.org/10.1145/1088149.1088161>.
- A. Gal. *Efficient bytecode verification and compilation in a virtual machine*. PhD thesis, Irvine, CA, USA, 2006. AAI3243940.
- A. Gal and M. Franz. Incremental dynamic code generation with trace trees. Technical report, Donald Bren School of Information and Computer Science, University of California, Irvine, 2006.
- A. Gal, B. Eich, M. Shaver, D. Anderson, D. Mandelin, M. R. Haghighat, B. Kaplan, G. Hoare, B. Zbarsky, J. Orendorff, J. Ruderman, E. W. Smith, R. Reitmaier, M. Bebenita, M. Chang, and M. Franz. Trace-based just-in-time type specialization for dynamic languages. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '09, pages 465–478, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-392-1. doi: <http://doi.acm.org/10.1145/1542476.1542528>. URL <http://doi.acm.org/10.1145/1542476.1542528>.
- A. Gill, J. Launchbury, and S. L. Peyton Jones. A short cut to deforestation. In *FPCA '93: Proceedings of the conference on Functional programming languages and computer architecture*, pages 223–232, New York, NY, USA, 1993. ACM. ISBN 0-89791-595-X. doi: <http://doi.acm.org/10.1145/165180.165214>. URL http://portal.acm.org/ft_gateway.cfm?id=165214&type=pdf&coll=GUIDE&dl=GUIDE&CFID=76299135&CFTOKEN=14374684.
- G. GMP. The gnu multiple precision arithmetic library. <http://gmplib.org>. Website, 2011.
- R. Gupta, D. A. Berson, and J. Z. Fang. Path profile guided partial dead code elimination using predication. In *Proceedings of the 1997 International Conference on Parallel Architectures and Compilation Techniques*, PACT '97, pages 102–, Washington, DC, USA, 1997. IEEE Computer Society. ISBN 0-8186-8090-3. URL <http://dl.acm.org/citation.cfm?id=522659.825652>.
- R. Gupta, D. A. Berson, and J. Z. Fang. Path profile guided partial redundancy elimination using speculation. In *Proceedings of the 1998 International Conference*

- on *Computer Languages*, ICCL '98, pages 230–, Washington, DC, USA, 1998. IEEE Computer Society. ISBN 0-8186-8454-2. URL <http://dl.acm.org/citation.cfm?id=857172.857261>.
- Hackage. Hackage. <http://hackage.haskell.org/packages/hackage.html>, 2010.
- R. E. Hank, W.-M. W. Hwu, and B. R. Rau. Region-based compilation: an introduction and motivation. In *Proceedings of the 28th annual international symposium on Microarchitecture*, MICRO 28, pages 158–168, Los Alamitos, CA, USA, 1995. IEEE Computer Society Press. ISBN 0-8186-7349-4. URL <http://portal.acm.org/citation.cfm?id=225160.225189>.
- J. E. Hannay, C. MacLeod, J. Singer, H. P. Langtangen, D. Pfahl, and G. Wilson. How do scientists develop and use scientific software? In *Proceedings of the 2009 ICSE Workshop on Software Engineering for Computational Science and Engineering*, SECSE '09, pages 1–8, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-1-4244-3737-5. doi: <http://dx.doi.org/10.1109/SECSE.2009.5069155>. URL <http://dx.doi.org/10.1109/SECSE.2009.5069155>.
- D. Hiniker, K. Hazelwood, and M. D. Smith. Improving region selection in dynamic optimization systems. In *Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 38, pages 141–154, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2440-0. doi: <http://dx.doi.org/10.1109/MICRO.2005.22>. URL <http://dx.doi.org/10.1109/MICRO.2005.22>.
- U. Hölzle and D. Ungar. Optimizing dynamically-dispatched calls with run-time type feedback. In *PLDI '94: Proceedings of the ACM SIGPLAN 1994 conference on Programming language design and implementation*, pages 326–336, New York, NY, USA, 1994. ACM. ISBN 0-89791-662-X. doi: <http://doi.acm.org/10.1145/178243.178478>. URL http://portal.acm.org/ft_gateway.cfm?id=178478&type=pdf&coll=GUIDE&dl=GUIDE&CFID=107683469&CFTOKEN=28437059.
- U. Hölzle, C. Chambers, and D. Ungar. Optimizing dynamically-typed object-oriented languages with polymorphic inline caches. In *ECOOP '91: Proceedings of the European Conference on Object-Oriented Programming*, pages 21–38, London, UK, 1991. Springer-Verlag. ISBN 3-540-54262-0.
- P. Hudak, J. Hughes, S. Peyton Jones, and P. Wadler. A history of haskell: being lazy with class. In *HOPL III: Proceedings of the third ACM SIGPLAN conference on History of programming languages*, pages 12–1–12–55, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-766-X. doi: <http://doi.acm.org/10.1145/1238844.1238856>. URL http://portal.acm.org/ft_gateway.cfm?id=1238856&type=mov&coll=GUIDE&dl=GUIDE&CFID=69896833&CFTOKEN=79663365.
- H. Inoue, H. Hayashizaki, P. Wu, and T. Nakatani. A trace-based java jit compiler retrofitted from a method-based compiler. In *CGO '11: Proceedings of the international symposium on Code generation and optimization*, 2011.

- R. Joshi, M. D. Bond, and C. Zilles. Targeted path profiling: Lower overhead path profiling for staged dynamic optimization systems. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, CGO '04, pages 239–, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2102-9. URL <http://dl.acm.org/citation.cfm?id=977395.977660>.
- G. Keller, M. M. Chakravarty, R. Leshchinskiy, S. Peyton Jones, and B. Lippmeier. Regular, shape-polymorphic, parallel arrays in haskell. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*, ICFP '10, pages 261–272, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-794-3. doi: <http://doi.acm.org/10.1145/1863543.1863582>. URL <http://doi.acm.org/10.1145/1863543.1863582>.
- J. Kim, W.-C. Hsu, and P.-C. Yew. Cobra: An adaptive runtime binary optimization framework for multithreaded applications. *Parallel Processing, International Conference on*, 0:25, 2007. ISSN 0190-3918. doi: <http://doi.ieeecomputersociety.org/10.1109/ICPP.2007.23>.
- J. Knoop, O. Rüthing, and B. Steffen. Optimal code motion: theory and practice. *ACM Trans. Program. Lang. Syst.*, 16(4):1117–1155, 1994. ISSN 0164-0925. doi: <http://doi.acm.org/10.1145/183432.183443>.
- T. Kotzmann, C. Wimmer, H. Mössenböck, T. Rodriguez, K. Russell, and D. Cox. Design of the java hotspot client compiler for java 6. *ACM Trans. Archit. Code Optim.*, 5:7:1–7:32, 2008. ISSN 1544-3566. doi: <http://doi.acm.org/10.1145/1369396.1370017>. URL <http://doi.acm.org/10.1145/1369396.1370017>.
- J. R. Larus. Whole program paths. In *Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, PLDI '99, pages 259–269, New York, NY, USA, 1999. ACM. ISBN 1-58113-094-5. doi: [10.1145/301618.301678](http://doi.acm.org/10.1145/301618.301678). URL <http://doi.acm.org/10.1145/301618.301678>.
- C. Lattner and V. Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *CGO '04: Proceedings of the international symposium on Code generation and optimization*, page 75, Washington, DC, USA, 2004. IEEE Computer Society. ISBN 0-7695-2102-9. URL http://portal.acm.org/ft_gateway.cfm?id=977673&type=pdf&coll=GUIDE&dl=GUIDE&CFID=91639383&CFTOKEN=59109600.
- D. X. Li, R. Ashok, and R. Hundt. Lightweight feedback-directed cross-module optimization. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, CGO '10, pages 53–61, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-635-9. doi: [10.1145/1772954.1772964](http://doi.acm.org/10.1145/1772954.1772964). URL <http://doi.acm.org/10.1145/1772954.1772964>.
- LLVM. The llvm compiler infrastructure. <http://llvm.org>. Website, 2011.

- J. Lu, H. Chen, R. Fu, W.-C. Hsu, B. Othmer, P.-C. Yew, and D.-Y. Chen. The performance of runtime data cache prefetching in a dynamic optimization system. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 36, pages 180–, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-2043-X. URL <http://portal.acm.org/citation.cfm?id=956417.956549>.
- J. Lu, H. Chen, P.-C. Yew, and W. chung Hsu. Design and implementation of a lightweight dynamic optimization system. *Journal of Instruction-Level Parallelism*, 6:2004, 2004.
- C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 190–200, New York, NY, USA, 2005. ACM. ISBN 1-59593-056-6. doi: <http://doi.acm.org/10.1145/1065010.1065034>. URL http://portal.acm.org/ft_gateway.cfm?id=1065034&type=pdf&coll=GUIDE&dl=GUIDE&CFID=74340410&CFTOKEN=80130600.
- S. Marlow and S. Peyton Jones. Making a fast curry: pushenter vs. evalapply for higher-order languages. *J. Funct. Program.*, 16(4-5):415–449, 2006. ISSN 0956-7968. doi: <http://dx.doi.org/10.1017/S0956796806005995>.
- S. Marlow, A. R. Yakushev, and S. Peyton Jones. Faster laziness using dynamic pointer tagging. In *ICFP '07: Proceedings of the 12th ACM SIGPLAN international conference on Functional programming*, pages 277–288, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-815-2. doi: <http://doi.acm.org/10.1145/1291151.1291194>. URL http://portal.acm.org/ft_gateway.cfm?id=1291194&type=pdf&coll=GUIDE&dl=GUIDE&CFID=89595348&CFTOKEN=43603062.
- D. G. Melski. *Interprocedural Path Profiling and the Interprocedural Express-Lane Transformation*. PhD thesis, University of Wisconsin-Madison, 2002.
- Z. Merali. Why scientific programming does not compute. In *Nature*, volume 467, pages 775–777. Nature, 2010. doi: [10.1038/467775a](https://doi.org/10.1038/467775a).
- W. Partain. The nofib benchmark suite of haskell programs. In *Proceedings of the 1992 Glasgow Workshop on Functional Programming*, pages 195–202, London, UK, 1993. Springer-Verlag. ISBN 3-540-19820-2.
- K. Pettis and R. C. Hansen. Profile guided code positioning. In *Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, PLDI '90, pages 16–27, New York, NY, USA, 1990. ACM. ISBN 0-89791-364-7. doi: [10.1145/93542.93550](https://doi.org/10.1145/93542.93550). URL <http://doi.acm.org/10.1145/93542.93550>.
- S. Peyton Jones and S. Marlow. The glasgow haskell compiler. <http://www.haskell.org/ghc>. Website, 2011.

- S. Peyton Jones, A. Tolmach, and T. Hoare. Playing by the rules: rewriting as a practical optimisation technique in ghc. In *Haskell Workshop*, 2001.
- S. L. Peyton Jones. *The Implementation of Functional Programming Languages (Prentice-Hall International Series in Computer Science)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1987. ISBN 013453333X.
- S. L. Peyton Jones. Implementing lazy functional languages on stock hardware: The spineless tagless g-machine. *J. Funct. Program.*, 2(2):127–202, 1992.
- S. L. Peyton Jones and J. Launchbury. Unboxed values as first class citizens in a non-strict functional language. In *Proceedings of the 5th ACM conference on Functional programming languages and computer architecture*, pages 636–666, New York, NY, USA, 1991. Springer-Verlag New York, Inc. ISBN 0-387-54396-1.
- S. L. Peyton Jones and D. R. Lester. *Implementing Functional Languages*. Prentice Hall, 1992.
- S. L. Peyton Jones and A. L. M. Santos. A transformation-based optimiser for haskell. *Sci. Comput. Program.*, 32(1-3):3–47, 1998. ISSN 0167-6423. doi: [http://dx.doi.org/10.1016/S0167-6423\(97\)00029-4](http://dx.doi.org/10.1016/S0167-6423(97)00029-4).
- N. Ramsey, J. a. Dias, and S. Peyton Jones. Hoopl: a modular, reusable library for dataflow analysis and transformation. In *Proceedings of the third ACM Haskell symposium on Haskell*, Haskell '10, pages 121–134, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0252-4. doi: <http://doi.acm.org/10.1145/1863523.1863539>. URL <http://doi.acm.org/10.1145/1863523.1863539>.
- Shootout. The computer language benchmarks game. <http://shootout.alioth.debian.org/>, 2009.
- SPEC. Spec benchmark suite. <http://www.spec.org/>, 2006.
- SPEC CPU Subcommittee. SPEC CPU2006 benchmark descriptions. In *Computer Architecture News*, volume 34. ACM SIGARCH newsletter, 2006.
- T. Suganuma, T. Yasue, and T. Nakatani. A region-based compilation technique for dynamic compilers. *ACM Trans. Program. Lang. Syst.*, 28:134–174, 2006. ISSN 0164-0925. doi: <http://doi.acm.org/10.1145/1111596.1111600>. URL <http://doi.acm.org/10.1145/1111596.1111600>.
- D. A. Terei and M. M. Chakravarty. An llvm backend for ghc. In *Haskell '10: Proceedings of the third ACM Haskell symposium on Haskell*, pages 109–120, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0252-4. doi: <http://doi.acm.org/10.1145/1863523.1863538>. URL http://portal.acm.org/ft_gateway.cfm?id=1863538&type=pdf&coll=GUIDE&dl=GUIDE&CFID=109508826&CFTOKEN=13519482.

- P. Wadler. Deforestation: transforming programs to eliminate trees. *Theor. Comput. Sci.*, 73(2):231–248, 1990. ISSN 0304-3975. doi: [http://dx.doi.org/10.1016/0304-3975\(90\)90147-A](http://dx.doi.org/10.1016/0304-3975(90)90147-A).
- D. Wakeling. The dynamic compilation of lazy functional programs. *J. Funct. Program.*, 8:61–81, 1998. ISSN 0956-7968. doi: <http://dx.doi.org/10.1017/S0956796897002955>. URL <http://dx.doi.org/10.1017/S0956796897002955>.
- R. C. Young. *Path-based Compilation*. PhD thesis, Harvard University, 1998.
- C. Zhao, Y. Wu, J. G. Steffan, and C. Amza. Lengthening traces to improve opportunities for dynamic optimization. In *12th Workshop on Interaction between Compilers and Computer Architectures*, Salt Lake City, UT, 2008.